HABILITATION À DIRIGER DES RECHERCHES

Présentée à

UNIVERSITÉ DE NICE SOPHIA-ANTIPOLIS - UFR SCIENCES
École Doctorale STIC

Spécialité INFORMATIQUE

par

DIDIER PARIGOT

Titre

CONTRIBUTION À LA PROGRAMMATION GÉNÉRATIVE

Soutenue publiquement le 27 novembre 2003
à l'INRIA devant le jury composé de

Président	Paul	FRANCHI-ZANNETTACCI	Université de Nice
Rapporteurs	Louis	FERAUD	Université de Paul Sabatier
	Jacques	MALENFANT	Université de Paris 6
	Christian	QUEINNEC	Université de Paris 6
Examinateurs	Daniel	DARDAILLER	Directeur du W3C Europe
	Martin	JOURDAN	MetaWare Technologies

Remerciements

Je voudrais remercier *Paul Franchi-Zannettacci* d'avoir accepté d'être le président de mon jury. Mais surtout, je lui suis reconnaissant de m'avoir lors de multiples discussions, influencé fortement ma manière d'aborder la recherche scientifique. En effet j'ai toujours apprécié chez *Paul* son esprit d'innovation, d'imagination, et son enthousiasme dans sa démarche scientifique. Je dois à *Paul* cette esprit d'ouverture et de réalisme qui (j'espère) caractérise ma démarche scientifique.

En me faisant l'honneur d'être rapporteurs de cette Habilitation à Diriger des Recherches (HDR), *Louis Feraud*, *Jacques Malenfant* et *Christain Quiennec* ont accepté d'évaluer mon travail de recherche qui implique leurs domaines excellences respectives.

En effet, *Louis Feraud* s'intéresse aux nouvelles technologies XML et cherche à promouvoir ses travaux de recherche (en particulier sur les grammaires attribuées) dans ce nouveau contexte. *Louis* pourra trouver ce même objectif dans mes travaux autour de la conception et l'implantation de l'outil SmartTools.

Pour sa part *Jacques Malenfant*, par ses compétences indiscutables en programmation par objets, pourra apprécier l'évolution récente de mes travaux de recherche vers cette nouvelle thématique pour moi.

Enfin, *Christian Queinnec*, qui a toujours été attentif à nos travaux sur les Grammaires Attribuées (avec *Martin Jourdan*, à Rocquencourt). *Christain* est issu principalement de la communauté, programmation fonctionnelle, qui, malheureusement, n'a jamais vraiment compris notre approche en terme de programmation fonctionnelle.

Ce jury n'aurait pas été complet sans la présence de *Martin Jourdan*, qui a été pendant plus de 10 ans successivement, mon encadreur de DEA, mon directeur de thèse et puis mon collègue de travail de tous les instants. En effet, pendant plusieurs années, nous avons construit ensemble un vrai travail d'équipe qui nous a permis de mener à bien des objectifs ambitieux, en particulier, notre outil de traitement de grammaires attribuées Fnc-2. Je voudrais, ici, lui dire un grand merci, pour cette collaboration intense qui n'est pas coutumière dans le monde de la recherche scientifique.

Enfin, je remercie *Daniel Dardailler* de sa présence, qui a su apprécier notre travail d'utilisation à tous les niveaux, des technologies XML. Je voudrai insister, à travers cette participation, sur mon souci quasiment constant dans ma démarche scientifique de rechercher des synergies entre divers domaines de recherche. Je suis convaincu que l'apport du W3C contribue fortement à l'établissement de ces synergies entre communautés scientifiques très variées.

Même s'il sera difficile à mes anciens de collègue de Paris d'être présents, je voudrais

dire, en particulier à *Etienne Duris*, *Gilles Roussel* et *Loic Correnson* le grand plaisir que j'ai eu à travailler avec eux et je garderai toujours un excellent souvenir de nos discussions scientifiques (avec souvent des crises de rire mémorables). Bien sûr, je n'oublie pas, mes anciens thésards *Catherine Julié*, *Bruno Marmol* et *Aziz Souah* avec qui j'ai eu aussi un grand plaisir à travailler.

Finalement, je voudrai finir par la super petite équipe "SmartTools" composée de *Carine Courbis*, *Pascal Degenne* et *Alexandre Fau*. Pendant trois ans, nous avons tous les quatre, travaillé ensemble avec un état d'esprit que je n'oublierai jamais. Je voudrais leur adresser toute ma reconnaissance pour leurs qualités techniques, scientifiques et surtout humaines.

Plan du manuscrit

Avant propos

Ce manuscrit se décompose en quatre parties. Après une courte introduction sur le bouleversement de l'informatique depuis quelques années, on énoncera les grands défis de l'informatique du logiciel pour les années à venir. Cette première partie me permettra de présenter, de mon point de vue, les futures pistes de recherche ou travaux qu'il faudra en priorité élaborer pour répondre à ces nouveaux défis. En particulier, on montera le rôle important de notre outil SMARTTOOLS en terme de démonstrateur de cette nouvelle approche. Cette partie correspond et argumente mon programme scientifique pour les années à avenir. La deuxième partie résume très rapidement mes travaux de recherche sur les grammaires attribuées qui influencent encore fortement mes activités de recherche. La troisième partie est composée de deux articles de présentation sur l'outil SMARTTOOLS qui décrivent et résument bien les grandes lignes directrices de ce logiciel. Enfin, la quatrième partie est un complément d'information sur mon dossier HDR (liste des projets de recherche, liste des publications, liste des sujets de DEA ou DESS encadrés, liste des thèses encadrées etc.)

Un large sous-ensemble de mes contributions (articles, réalisation de logiciels, documentations) sont accessible par ma page Web,
http://www-sop.inria.fr/oasis/personnel/Didier.Parigot/.

Partie I : Synthèses et perspectives de mes travaux de recherche (page 3)
Cette partie décrit rapidement mes travaux de recherche depuis trois ans centrés autour de la conception et du développement de l'outil SMARTTOOLS. Cette présentation donne un aperçu précis sur mes directives de recherche futures.

Partie II : Travaux sur les Grammaires Attribuées (page 33)
Ce chapitre résume l'ensemble de mes travaux théoriques sur les grammaires attribuées [103]. Cette présentation donne juste un aperçu très rapide de plus 10 ans de travaux de recherche sur les grammaires attribuées (GAs). Pour bien comprendre l'ampleur de mes travaux sur ce thème, la réalisation d'un système de traitement de grammaires attribuées, appelé FNC-2 le lecteur pourra se référer au manuel d'utilisation de FNC-2 [88]. Avec ce manuel, le lecteur comprendra les efforts de dévelop-

pement que cela a entraînés.

Le manuel d'utilisation de FNC-2 est accessible à :
ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/f2manual.ps.gz.

Partie III : Deux chapitres sur l'outil SMARTTOOLS

Les deux chapitres suivants (textes de deux articles [53, 54]) décrivent l'outil SMARTTOOLS. Le lecteur pourra aussi se référer à la documentation sur SMARTTOOLS (http://www-sop.inria.fr/oasis/SmartTools/st4up).

Chapitre 3 : Présentation générale de SMARTTOOLS (page 69)

Ce chapitre donne une vue d'ensemble de l'outil SMARTTOOLS [53], cible de nos expérimentations. Cette vue permet d'appréhender les besoins et rouages de l'outil mais aussi son évolution. Elle montre comment l'approche par génération de code et l'usage de technologies standardisées avantagent le développement et l'évolution d'un tel logiciel.

Chapitre 4 : Architecture par composants (page 93)

Ce chapitre explique pourquoi nous avons préféré créer notre propre modèle de composants plutôt qu'en utiliser un existant tel que CORBA ou EJB. Ce modèle est décrit [52] ainsi que les principaux composants de l'outil et sa mise en œuvre. Les avantages principaux de nos composants sont leur adéquation aux besoins, leur extensibilité de services et leur capacité à être exportés vers d'autres technologies [164].

Partie IV : Le dossier (page 115)

Cette partie donne des informations complémentaires sur mon dossier d'HDR.

Table des matières

Table des figures

Première partie

Synthèses et perpectives des travaux de recherche

Chapitre 1

Introduction

1.1 Les nouveaux défis pour le développement logiciel

La qualité du logiciel et sa capacité à évoluer, ainsi que la rapidité du développement, sont des soucis majeurs pour les industriels. Un logiciel bien conçu doit pouvoir s'adapter rapidement aux demandes des clients et aux nouvelles technologies pour pouvoir lutter contre la concurrence et les nouvelles technologies. Il doit aussi être capable d'échanger des données très variées avec d'autres applications, particulièrement depuis l'avènement d'Internet. Puis il doit pouvoir s'exécuter sur divers supports matériels, de manière répartie ou distribuée avec une très grande souplesse d'utilisation. Il doit aussi pouvoir être construit et développé par assemblage de composants sur l'étagère, pour assurer une meilleure évolution dans le cycle de vie et une ouverture vers d'autre application. Enfin, l'utilisation de l'informatique usuellement confinée aux domaines scientifiques ou techniques s'est très largement démocratisée, depuis quelque années, vers des utilisateurs grand public, grâce en particulier au succès de l'Internet ou à l'informatique monade (téléphones portables, PDA etc). Ces utilisateurs ont des besoins, des connaissances et des domaines d'activité, extrêmement variés et différents qui évoluent très rapidement. De plus, la pression du marché impose des temps de développement de logiciel plus courts et des coûts plus faibles.

Il faut bien admettre que l'on est passé d'une utilisation de l'informatique très centralisée (centre de calcul), en passant par une informatique pour l'ingénieur ou pour le technicien spécialisé dans les années 80-90, époque de la décentralisation de l'informatique, vers une informatique grand public aussi facile d'utilisation et répandue que l'électricité.

Cette évolution bouleverse très fortement la manière de concevoir et de réaliser les logiciels dans le futur. De nos jours, il n'est plus possible de développer une solution logicielle

sans l'utilisation de composants réalisés par d'autres partenaires. Le temps où un industriel était le maître d'œuvre sur toute la chaîne d'une solution informatique est définitivement résolu. Les technologies propriétaires qui étaient la panacée des débuts de l'informatique disparaissent face aux efforts de standardisation des consortiums comme le W3C (*World Wide Web Consortium*)[11] pour l'Internet ou le développement du logiciel avec l'OMG (*Object Management Group*) [16]. Les exigences vis-à-vis des logiciels ont aussi été modifiées à cause des disparités de connaissances des utilisateurs et des programmeurs, des contraintes de temps et de financement, et des nécessités d'adaptation rapide aux besoins du marché. Les logiciels doivent aussi satisfaire les exigences suivantes :

- conviviaux grâce à une interface utilisateur interactive ;
- faciles à utiliser avec peu de compétences informatiques et basés sur des techniques bien connues ou des standards ;
- **ouverts** grâce à un format d'échange de données standard utilisé pour communiquer entre les composants et avec les applications externes ;
- **adaptables** rapidement à leur environnement grâce à un développement s'appuyant sur une implantation modulaire et flexible, basée sur des composants génériques et réutilisables.

Pour prendre en compte ces bouleversements, de nouvelles techniques de développement ont émergées. Tout d'abord, il y a eu la Programmation par objets avec les notions d'encapsulation et d'héritage propices à la modularité, la réutilisation et l'extensibilité du code. Mais ce style de programmation est apparu insuffisant pour prendre en compte des préoccupations transversales aux classes.

Programmation par aspects

Pour pallier ce problème, la programmation dite par aspects [32, 71, 100] (AOP - *Aspect-Oriented programming*) a, en particulier, vu le jour permettant de gérer, de manière modulaire, ces préoccupations en les séparant du code de base (*separation of concerns*) ; la plus connue des implémentations étant *AspectJ* [4, 101] de Kiczales. Il est ainsi très facile d'étendre le code de base avec de nouvelles fonctionnalités sans le modifier directement (technique non invasive). Cette technique jeune soulève de nombreuses questions : où tisser les aspects ? Comment composer plusieurs aspects ? Quelle technique d'implémentation choisir entre la transformation de programme, la réflexivité ou la génération de code adapté ? On peut aussi citer les approches par programmation par sujets [78] ou par méta-programmation [113] ou par programmation adaptative [110]. J'expliquerai un peu plus tard pourquoi je m'intéresse plus précisément à la programmation par aspects pour cette démarche par séparation des préoccupations, (voir section 1.5).

Programmation par composants

L'approche objet a aussi une granularité très fine, peu adaptée aux systèmes complexes. Le concept de composant [116] a été introduit afin d'encapsuler plusieurs objets proposant des services et de pouvoir facilement y associer du code non-fonctionnel tel que la

communication entre composants, la persistance ou la politique de sécurité. Avec cette approche par composants, il est facile de déployer et même de répartir une application. Il est aussi possible de construire une application uniquement par assemblage de composants, ce qui devrait réduire les coûts et les temps de développement. Quatre principaux modèles de composants ont émergé : les EJBs (*Entreprise JavaBeans*) [10] de Sun, CCM (*CORBA Component Model*) [7] de l'OMG (*Object Management Group*), DCOM (*Distributed Component Object Model*) et .NET de Microsoft et les *Web Services* du W3C (*World Wide Web Consortium*). Avec l'avènement des *Web Services*, cette possibilité de création d'applications par assemblage de composants attire de nombreux industriels. Mais il reste encore de nombreuses interrogations : comment faire communiquer des composants de modèles différents ? Comment découvrir les composants ayant les services et la qualité de service (QoS) souhaités ? Quelle sécurité adopter pour les transactions ?, Comment rendre adaptable les composants ?. Sur le thème des composants il existe de nombreux travaux de recherche [153] qui émanent de plusieurs communautés de recherche différentes (système, application distribuée [117], architecture logicielle avec les travaux sur les ADLs [40, 58], Web sémantique avec les Web Services [154]).

Intégrité des données

Avec l'effervescence liée à la naissance du Web, il y a eu une volonté de standardiser les langages et protocoles, ce qui a conduit à la création du W3C [11]. Ce consortium a proposé un nouveau format de données, XML (*Extensible Markup Language*), issu de SGML (*Standard Generalized Markup Language*) afin de simplifier et d'uniformiser les échanges d'informations entre applications, indépendamment de tout langage et plate-forme. Ce consortium a ensuite élaboré des spécifications[1] de langages, de protocoles et d'APIs (*Application Programming Interface*) tels que le langage XSLT (*Extensible Stylesheet Language Transformation*) pour effectuer des transformations, le protocole SOAP (*Simple Object Access Protocol*) pour échanger des messages ou l'API DOM (*Document Object Model*) pour manipuler les documents sous forme d'arbres. Par exemple, le protocole SOAP est utilisé par les Web Services. Utiliser des technologies standardisées et pour lesquelles il existe de nombreux outils simplifie le développement, l'évolution et rend les applications ouvertes. On peut aussi mentionner les travaux sur les base de données semi-structurées [20] depuis l'avènement d'XML.

Programmation par modèles ou l'approche par méta-modélisation

Ces bouleversements ont aussi été pris en compte au niveau des spécifications des logiciels, avec l'apparition de nouvelles méthodes d'analyse et de conception à base de modèles et à production de code. La plus célèbre et récente de ces méthodes de modélisation de systèmes à objets est UML (*Unified Modeling Language*) [18] de l'OMG qui réunit et

[1] Les spécifications du W3C (XML et DTD, DOM, XSL et XSLT, XML Schema, BML, MathML, SVG, XPath, XHTML, SOAP, WSDL, etc.) sont accessibles sur le site http://www.w3c.org.

unifie les notations et les sémantiques des méthodes Booch de Grady Booch, OMT (*Object Modeling Technique*) de Jim Rumbaugh et OOSE (*Object-Oriented Software Engineering*) de Ivar Jacobson. Avec ce langage graphique, les différentes dimensions d'un logiciel complexe peuvent être spécifiées, assemblées, visualisées et documentées. Ces différentes modélisations UML sont ensuite utilisées pour générer automatiquement la trame du code du logiciel cible par exemple avec l'Rationale Rose d'IBM. Récemment, l'OMG a introduit une nouvelle approche d'écriture de spécifications et de développement d'applications, nommée MDA[2] (*Model-Driven Architecture*) [37, 77, 156, 118, 38]. Cette approche prône l'utilisation de modèles indépendants de toute plate-forme et technologie (PIM - *Plateform-Independant Model*) qui sont ensuite transformés vers un ou plusieurs modèles de plate-forme spécifique (PSM - *Platform-Specific Model*). La partie métier (PIM) est ainsi séparée de la partie technologie cible (PSM) et le passage du PIM vers un ou plusieurs PSMs s'effectue par des règles de transformation. Cette approche par abstraction permet de mieux se concentrer sur la partie «intelligente» et la rend pérenne car utilisable avec les technologies futures. L'un des intérêts (résumé dans la figure 1.1) de cette approche est de permettre l'évolution des applications produites en faisant évoluer les différents générateurs ou les transformations de modèles.

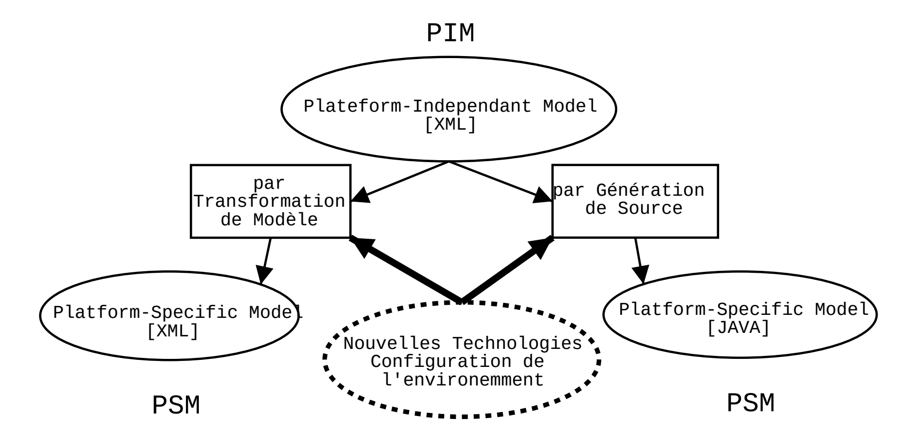


FIG. 1.1 – L'approche Model-Driven Architecture (MDA)

1.2 Une nouvelle approche pour le développement logiciel

Un changement radical des méthodologies de conception et de développement d'application est nécessaire pour aisément prendre en compte ces nouvelles approches de développement telles que la programmation par aspects, les composants et la stratégie de développement MDA.

Nous allons énoncer les bases d'une nouvelle manière de programmer où ces approches sont automatiquement intégrées aux spécifications (modèles abstraits) de l'application lors

[2] Des informations de cette approche sont disponibles à http://www.omg.org/mda ou http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda/

de phases de génération de code source [57]. Cette idée a été appliquée, à différents niveaux, lors de la réalisation de l'outil SMARTTOOLS, aussi bien pour la représentation de données ou d'environnements interactifs que pour les traitements sémantiques ou l'architecture par composants. La figure 1.2 schématise l'approche prise pour la conception de SMARTTOOLS en montrant les différents PIMs et PSMs mise en œuvre .

Cette approche défend et montre, de manière pragmatique, les apports de ces différentes approches et comment les combiner ensemble. Nos objectifs avec cet outil s'inscrivent parfaitement dans cette nouvelle problématique, aussi bien pour sa réalisation que pour les environnements qu'il produit ; son but étant d'aider à la création d'outils tels que des éditeurs, des afficheurs (*pretty-printers*), des analyseurs syntaxiques (*parsers*) ou des traitements sémantiques (analyses, transformations) pour les langages de programmation ou métiers. Plus précisément, à partir d'une description d'un langage ou modèle de données (DTD ou XML Schema), il génère un environnement minimal doté d'un éditeur guidé par la syntaxe, d'un ensemble de fichiers Java facilitant l'écriture de traitements sémantiques, d'afficheurs génériques et d'un analyseur syntaxique XML ayant les fonctions de construction d'arbres de ce langage. Cet environnement peut, ensuite, être enrichi par d'autres outils.

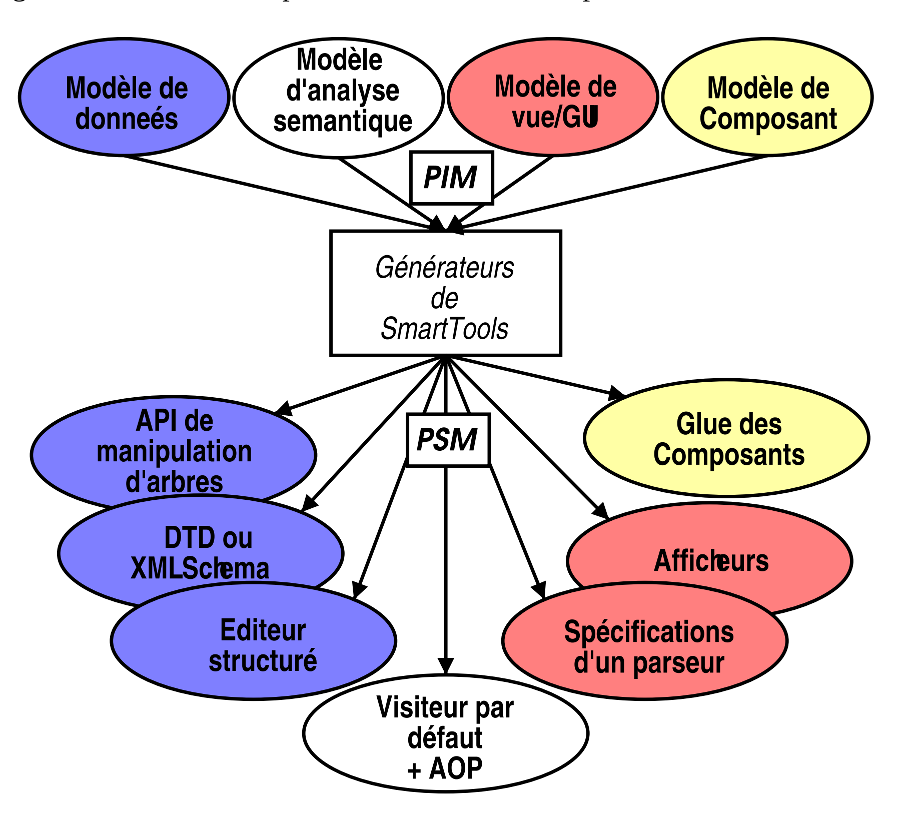


FIG. 1.2 – Schéma fonctionnel de SMARTTOOLS

L'originalité et l'innovation de notre approche, dans le contexte de l'outil, peuvent se synthétiser en les points suivants :

1. Fournir une interface utilisateur construite à partir d'un modèle (structure d'arbre).

L'innovation de notre approche consiste à traiter tous les aspects d'affichage, y compris l'interface utilisateur, selon le même modèle. Il se dégage ainsi une approche homogène et uniforme, ayant un fort potentiel de réutilisation tant pour SMARTTOOLS que pour les outils produits. Un autre avantage important est que les techniques utilisées permettent d'exporter les vues graphiques vers d'autres supports dont le Web (navigateurs).

2. Accepter et utiliser des formats non propriétaires définis par le W3C et profiter ainsi des nombreux développements réalisés autour de XML. De cette manière, le coût et le temps de développement de l'outil peuvent être fortement réduits. Notre innovation consiste à proposer des traitements sémantiques sur des documents XML, en utilisant une méthodologie de programmation basée sur le patron de conception visiteur (*visitor design pattern*) [73, 126, 127], issu de la programmation par objets.
3. Proposer une programmation par aspects [32, 98, 101] au-dessus de la technique des visiteurs ne requérant pas de transformation de code. Cette approche dynamique a l'intérêt d'être beaucoup plus simple dans sa mise en œuvre que les approches plus classiques et généralistes [110]. Mais surtout, elle aura certainement un grand intérêt dans le cadre d'applications Web pour traiter les problèmes de reconfiguration, d'adaptation et de sécurité des composants.
4. Posséder une architecture logicielle modulaire [27] (avec des composants indépendants) et extensible pour assurer une bonne évolution de l'outil. Ainsi, il est facile d'ajouter de nouvelles fonctionnalités (importation de nouveaux composants) ou de partager notre savoir-faire (exportation de nos composants) avec d'autres plates-formes. Chaque environnement produit est aussi un composant indépendant.

De plus notre approche est auto-utilisée pour son propre développement. Ainsi, les techniques proposées sont directement testées. Par exemple, tous nos langages internes ont été développés grâce à l'outil. Chaque environnement produit réutilise les composants existants. Pour un aperçu de cet outil, il suffit de regarder la documentation[3] en cours d'élaboration ou le site de SMARTTOOLS [4]. Dans la figure 1.3, nous montrons l'ensemble des PIMs de SMARTTOOLS qui composent l'environnement pour un langage.

Le lien conducteur de cette approche prise dans SMARTTOOLS est certainement la programmation générative qui fédère toutes ces différentes technologies, simplifie leur usage et facilite les évolutions. Cette génération, à partir de modèles, correspond à un sous-ensemble de l'approche MDA. La réalisation et la conception de SMARTTOOLS illustre bien cette nouvelle approche du développement d'applications complexes ayant comme clé de voûte la génération de code à partir de modèles ou de spécifications abstraites Cette approche orientée modèle peut, dans une certaine manière, être comparée à l'approche MDA [37, 77, 156] de l'OMG. Les intérêts principaux d'une telle approche sont une meilleure qualité logicielle due à la mise en exergue du code intelligent et à la génération de code, une facilité de portabilité vers d'autres plates-formes et une prise en compte rapide des évolutions technologiques (par création ou modification des règles de transformation).

[3] http://www-sop.inria.fr/oasis/SmartTools/st4up/
[4] http://www-sop.inria.fr/oasis/SmartTools/

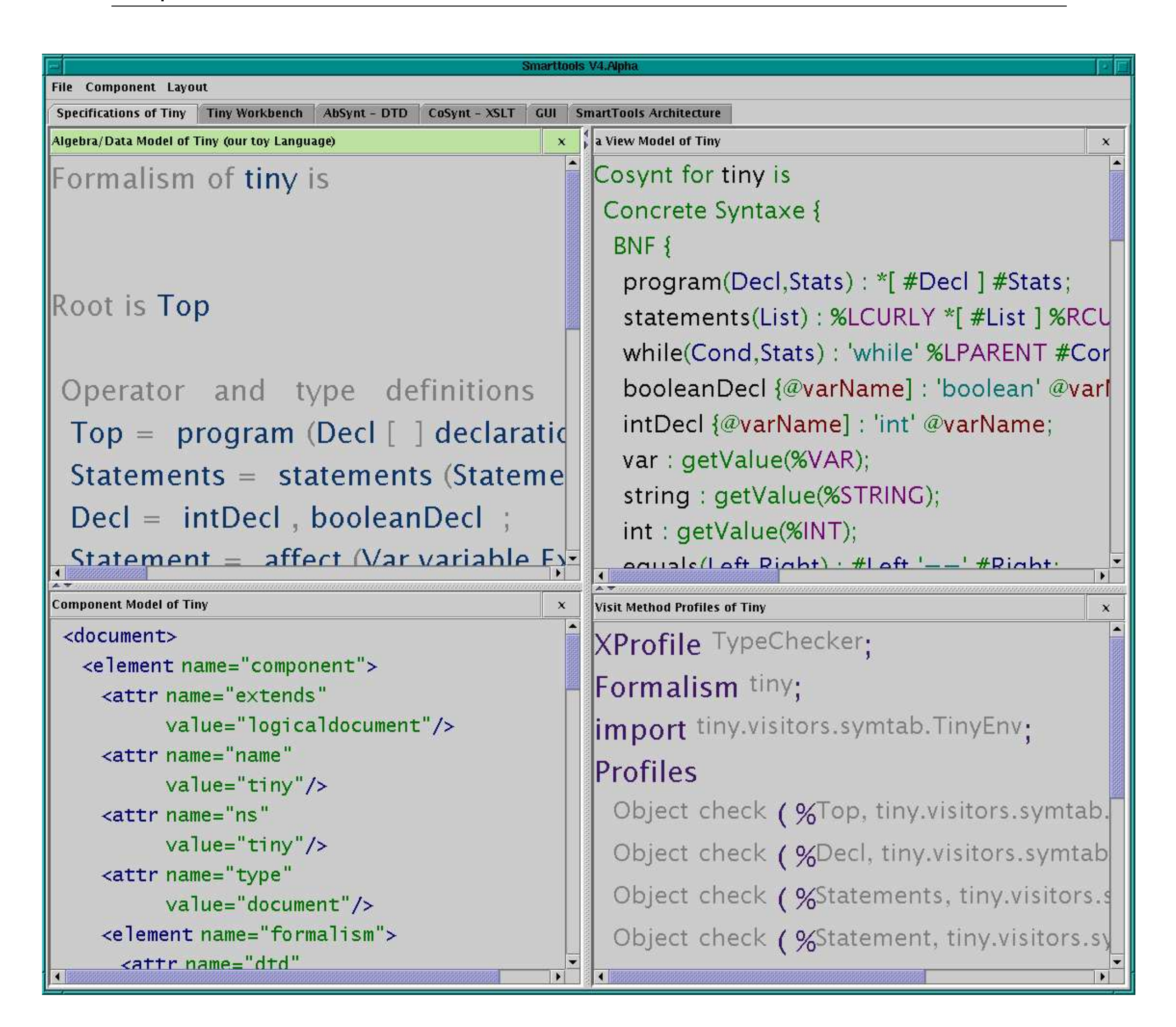


FIG. 1.3 – Exemple d'interface utilisateur

Lors de la construction de SMARTTOOLS, cette approche, selon notre vision, a été adoptée non seulement pour l'architecture, mais aussi pour la définition de structure des langages, pour la création d'afficheurs, d'analyseurs syntaxiques, de vues et de l'interface graphique, et pour la configuration des visiteurs. Nous montrons, ci-dessous, comment elle a été appliquée à ces différentes parties. Précisément sur chaque partie, on explicitera le modèle PIM, le modèle PSM et le type de la transformation employée (parfois de manière implicite). Finalement, nous proposons à travers notre outil SMARTTOOLS un nouveau style de programmation que l'on peut comparer à une notion de fabrique ou usine d'application. En effet, SMARTTOOLS propose une chaîne de montage qui permet de produire rapidement une famille d'application orientées langage. Cette chaîne de montage est basée sur les quatre concepts suivants que nous allons détailler par la suite :

- ouverture vers des formats neutres pour la description des structures de données ;
- environnement interactif et graphique ;
- une programmation par séparation des préoccupations fortement basée sur une approche de programmation générative ;
- une programmation par composants indépendante d'une technologie à composants particulière.

1.2.1 Passerelle vers les formalismes du W3C (DTD et XML Schema)

Adopter des formats de données standardisés facilite l'échange d'informations entre logiciels. Le W3C donne la possibilité aux concepteurs de décrire les structures de données échangées en utilisant les formalismes DTD (*Document Type Definition*) ou XML Schema. Ainsi ils définissent des langages dits métiers (par opposition aux langages de programmation) très variés et liés à un domaine d'application : télécommunications, mais aussi finance, assurances, transports, etc. Les techniques liées aux langages de programmation peuvent être employées pour les langages métiers, d'autant plus que ces derniers ont souvent une syntaxe et une sémantique plus simples. Mais les concepteurs et les utilisateurs de langages métiers n'ont pas forcément de compétences approfondies sur les techniques issues de la programmation (analyse, compilation, interprétation, etc). Il y a donc un besoin d'outils pour faciliter l'utilisation de ces techniques. De plus, de telles applications (liées à l'Internet) nécessitent un développement rapide, des possibilités d'intégration, une utilisation facile et un affichage multi-supports tel que l'écran d'un PDA - *Personal Digital Assistant* -, d'un téléphone mobile, un éditeur de texte ou un navigateur Internet.

Le langage ABSYNT (PIM) (Cf. la figure 1.4 un exemple complet pour notre petit langage de démonstration TINY), indépendant de tout langage de programmation et adapté à nos besoins (fils optionnels ou opérateur d'arité non bornée), a été créé afin d'obtenir une définition, de haut niveau, des syntaxes abstraites des langages. A partir d'une telle définition de langage, on génère des classes Java (la cible ou PSM), construites au-dessus de l'API DOM ; ces classes serviront pour représenter les nœuds de ses arbres. Ainsi les concepteurs de langages se concentrent uniquement sur cette définition, non sur l'implémentation du langage. De plus, il est facile, par simple modification du générateur, d'adopter une autre API d'arbres. La transformation du PIM en PSM est de type génération de code et le PSM correspond à un raffinement du PSM API DOM. Dans le futur, ce dernier pourrait être remplacé par d'autres PSMs.

Pour faciliter la création de langages métiers dédiés au Web, toute spécification ABSYNT (PIM) peut être traduite en DTD ou en XML Schema (PSM)[5], par génération de code. Les concepteurs de tels langages, en utilisant SMARTTOOLS, peuvent ainsi profiter d'une syntaxe de description de documents (ABSYNT) beaucoup plus concise donc plus facile à appréhender et de ses outils pour la visualisation et d'outils pour les traitements sémantiques.

Afin que l'outil soit ouvert et ait d'autres champs d'applications, les passerelles inverses ont aussi été établies. Cette abstraction ou conversion est assez complexe à spécifier car le

[5]Ces PSM sont liés aux technologies XML.

```
Formalism of tiny is

Root is Top;

Operator and type definitions {
  Top = program(Decl[] declarations, Statements statements);
  Statements = statements(Statement[] statementList);
  Decl = intDecl(),
         booleanDecl();
  Statement =
      affect(Var variable, Exp value),
      while(ConditionExp cond, Statements statements),
      if(ConditionExp cond, Statements statementsThen, Statements statementsElse),
      println(StringOrVar[] stringOrVars);
  StringOrVar =
      var,
      string as java.lang.String;
  ConditionOp =
      equal(ArithmeticExp left, ArithmeticExp right),
      notEqual(ArithmeticExp left, ArithmeticExp right);
  ConditionExp =
      %ConditionOp,
      true(),
      false(),
      var;
  ArithmeticOp =
      plus(ArithmeticExp left, ArithmeticExp right),
      minus(ArithmeticExp left, ArithmeticExp right),
      mult(ArithmeticExp left, ArithmeticExp right),
      div(ArithmeticExp left, ArithmeticExp right);
  ArithmeticExp =
      %ArithmeticOp,
      int as java.lang.String,
      var as java.lang.String;
  Exp =
      %ArithmeticOp,
      var,
      int,
      true,
      false;
  Var = var;
}

Attribute definitions {
REQUIRED varName as java.lang.String in intDecl, booleanDecl; // pb avec %Decl
}
```

FIG. 1.4 – Exemple de syntaxe abstraite en ABSYNT pour un petit langage TINY

modèle de départ (XML Schema XML) est plus riche et peut donc comporter quelques pertes d'information (les contraintes de structure sont plus strictes pour les langages de programmation). En effet, notre problème est de définir une correspondance exacte entre les éléments du document XML et des classes (objets) Java qui vont représenter l'arbre de syntaxe abstraite. Avec les DTDs, il est possible de définir des éléments avec des parties droites pouvant être très complexes (une expression régulière). Pour les XML Schema , par

construction, on peut mieux structurer le document (le langage propose des éléments pour cela) mais rien n'empêche de définir un élément sans passer par ces niveaux intermédiaires.

On retrouve cet effort d'interprétation (de transformation) des formalismes du W3C en terme de syntaxe abstrait dans les travaux de recherche de [149, 34, 81] pour les langages de programmation et pour les bases de données [20]. Mais notre souci a été d'assurer la transformation inverse (sérialisation). Plus précisément, le programme en format XML doit toujours être valide par rapport à la DTD ou le Schéma XML associé. Dans notre cadre, il était donc impossible d'inventer des nouveaux types (pour mieux structurer le document).

Tout cela est résumé dans la figure 1.5 pour la partie définition des structures de données.

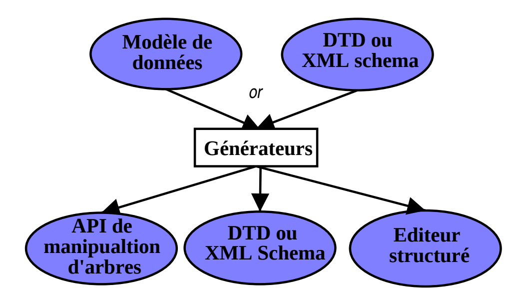


FIG. 1.5 – Outils générés à partir du modèle de données

Nos efforts pour accepter les formalismes du W3C sont certes motivés par notre souci d'élargir le champ d'applications de SMARTTOOLS mais aussi par notre volonté de faciliter son utilisation. De cette manière, les utilisateurs ne sont pas contraints à apprendre nos langages internes. L'outil accepte, en entrée, aussi bien notre propre langage de définition d'AST (*Abstract Syntax Tree*) qu'une DTD XML Schema XML.

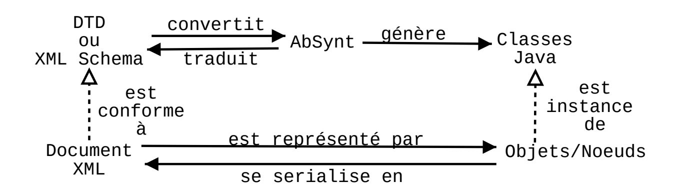


FIG. 1.6 – Passerelle entre les langages et les documents

En suivant la même approche, établir une passerelle vers la méta-modélisation (UML, le concept de MOF - *Meta Object Facility*), semble être aussi très naturel. Par exemple dans [37] l'auteur effectue un parallèle entre les divers niveaux (quatre niveaux pour la modélisation) de spécification introduits dans cette communauté et les divers niveaux d'abstraction allant du programme, de la grammaire du langage sous-jacent et enfin du méta-langage utilisé pour définir cette grammaire.

1.2.2 Édition structurée et environnement interactif

Dans le domaine des éditeurs structurés [31, 102, 140], les langages métiers définis à l'aide des formalismes XML sont certainement de meilleurs candidats que les langages dits de programmation, où les éditeurs professionnels et spécialisés (*Eclipse*, *Sun ONE* anciennement *Forte*, *Visual Studio*™, *JBuilder*™, *Visual Age*™, etc) sont des concurrents manifestes à l'approche générique. Mais il est important, même pour ces langages métiers [161] beaucoup moins exigeants en terme d'édition libre (possibilité d'écrire du code à la volée), de proposer des outils d'affichage ouverts et extensibles à de nouvelles bibliothèques de composants graphiques telle que la bibliothèque `Swing`.

Mais surtout, donner des moyens pour visualiser ou manipuler la structure de données (la structure logique de l'application) sous-jacent permet une meilleur organisation de l'application. Il est important que les problèmes de représentation, de mémorisation, de visualisation n'influencent en aucune manière, la structure logique de l'application. D'une autre manière il est important d'assurer une meilleur séparation entre les préoccupations en offrant ces services puissants (environnement interactif). Il est important que ces services soient adaptables et ouverts en utilisant le plus possible des technologies standards pour éviter des développements lourds et importants.

Nous allons montrer qu'il est relativement simple de construire au-dessus de l'outil de transformation XSLT (voir section 3.5), un mécanisme d'affichage de vues avec les contraintes particulières liées à l'édition structurée. En effet, il faut assurer une cohérence entre l'état de la vue et la structure de données. Cela impose de concevoir des transformations réversibles ou symétriques. Cette approche (avec BML - *Bean Markup Language*) rend aussi l'exportation aisée des vues graphiques à travers le réseau. L'utilisation de due format BML permet un transfert sur le réseau, d'assemblage complexe d'objets graphiques.

L'utilisation de diverses familles technologiques – la bibliothèque `Swing` pour le graphisme, l'outil XSLT pour les transformations, BML pour la *sérialisation* et CSS (*Cascading Style Sheet*) pour l'application de styles – apporte une solution efficace à faible coût de développement et de maintenance. Développer nos propres algorithmes de placement d'objets graphiques, langages de transformation et moteurs, nous aurait demandé beaucoup d'énergie et de temps pour un résultat qui, bien que peut-être plus efficace (car conçus pour et non adaptés), aurait été non évolutif et non ouvert. Notre solution profite ainsi des avantages de chaque technologie et surtout de leurs futures évolutions. L'utilisation des technologies XML est un atout indéniable de notre approche.

De plus, les interfaces graphiques des applications qui, sont susceptibles d'être exécutées dans des environnements ou supports très variés, doivent de plus en plus être modulables ou reconfigurables en fonction du support de visualisation ou de manipulation. Nous pensons que notre approche est un premier pas pour garantir cette forte possibilité de reconfiguration de l'interface d'utilisateur, grâce à ce format neutre (BML) de représentation des objets à visualiser. Nous avons déjà effectué quelques expériences dans ce sens (voir le chapitre 3) en exportant facilement nos vues graphiques sur des navigateurs Web à travers le réseau.

Pour montrer que ces outils d'édition ne sont forcément pas liés à des langages de programmation, il suffit de regarder comment l'interface utilisateur (GUI) de SmartTools

a été conçue. Dans la figure 1.7, on peut voir un exemple de notre langage de description d'interface utilisateur. L'interface utilisateur est une visualisation de cette description (un arbre XML). En effet, pour l'interface utilisateur, nous avons appliqué la même approche que pour la visualisation des programmes . L'interface n'est qu'une vue particulière (avec des objets graphiques très spécifiques à une interface utilisateur) d'un arbre qui n'a plus rien à voir avec la visualisation d'un langage de programmation. Cela montre aussi que notre approche ne doit pas être limitée à la création d'environnements pour les langages de programmation mais à toute application ou la structure de données peut être formalisée sous forme d'arbre.

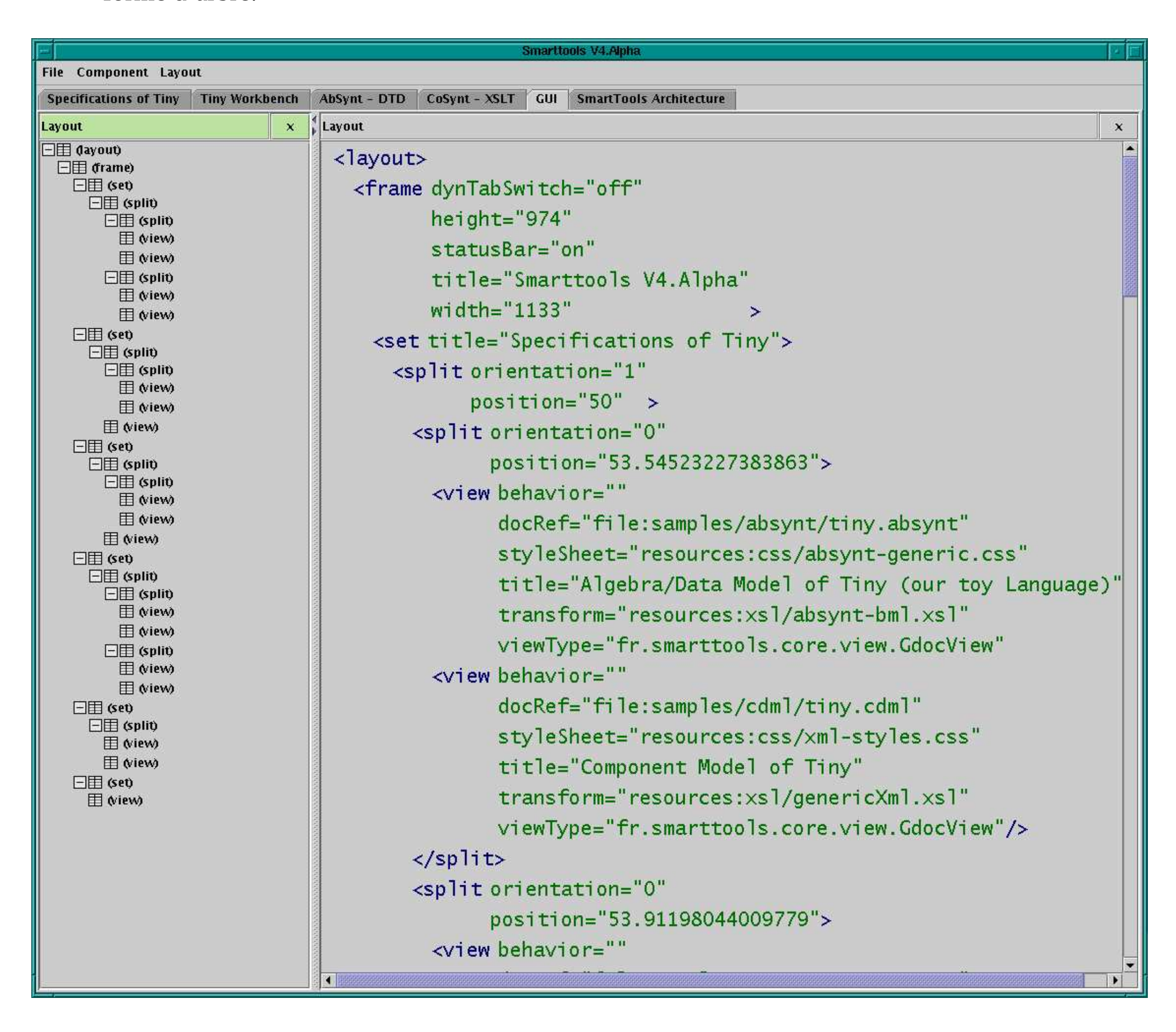


FIG. 1.7: Langage LML de description d'interface utilisateur

Pour que le concepteur d'une application ne se focalise que sur la définition de la structure logique de son application (syntaxe abstraite), nous avons défini des outils de trans-

formation (de visualisation) qui prennent comme base de départ, la syntaxe abstraite. Nous avons ainsi défini un langage (voir la figure 1.8), CoSynt qui permet de spécifié, à partir d'une syntaxe abstraite, une syntaxe concrète et plusieurs vues graphiques (mise en forme) de cette dernière. Le langage CoSynt, indépendant de toute implémentation, sépare les différentes préoccupations relatives à la syntaxe concrète d'un langage et à l'affichage ; la partie syntaxe concrète se compose de la BNF, de la valeur du *lookahead* et des expressions régulières du lexeur, et la partie affichage de noms de style, de règles de transformation de l'arbre de syntaxe concrète en arbre de boîte, et d'objets graphiques à utiliser selon la sortie souhaitée.

Grâce à la spécification AbSynt (notre langage de description de syntaxe abstraite) d'un langage et à une de ses spécifications CoSynt (l'ensemble formant le PIM), l'outil peut générer soit une spécification d'analyseur syntaxique de type ANTLR, soit un ou plusieurs afficheurs écrits à l'aide de transformation XSLT.

Avec la feuille de style générée de l'afficheur, le processeur XSLT transforme un arbre de syntaxe abstraite (la feuille de style et l'arbre instance du PIM) en un arbre de boîtes (*beans*) ou un fichier texte (PSM). L'arbre de boîtes peut ensuite être personnalisé (raffiné) par une feuille de style CSS. Les vues graphiques et l'interface graphique sont obtenues selon le même procédé. De manière conceptuelle (voir la figure 1.9), l'arbre de syntaxe abstraite est transformé en un arbre de syntaxe concrète puis d'un arbre de boîtes et enfin d'un arbre de boîtes enrichies d'attributs de style, alors que dans l'implémentation il n'existe qu'une seule transformation (XSLT) qui génère directement le dernier modèle ; l'arbre de syntaxe abstraite étant le PIM et l'arbre de boîtes enrichies d'attributs de style le PSM, obtenu par raffinements successifs de l'arbre de syntaxe concrète et de l'arbre de boîtes. Mais il est important de noter qu'en terme d'implantation pour ce langage de transformation, nous nous sommes fortement basés sur les outils ou technologies standards. la figure 1.10 résume l'utilisation des technologies standards pour la mise en œuvre de CoSynt.

1.2.3 Les outils sémantiques

Le succès grandissant de la notion de patrons de conception [73] montre que les concepts de programmation par objets ne sont pas suffisants et que chaque type d'application ou problématique demande des solutions appropriées [122]. En particulier, le patron visiteur, à la base de nos outils sémantiques, a suscité un ensemble de travaux de recherche [127, 79, 120, 104, 80, 72], qui ont tous comme objectif de trouver le meilleur compromis entre la lisibilité et l'efficacité du programme. L'un des autres soucis de ce patron est la composition de visiteurs [104].

Certaines similitudes sur cette problématique avaient déjà été remarquées dans [46] mais pour des familles de technologies différentes (grammaires attribuées [79, 145], programmation polytypique [83] et programmation adaptative [128, 110]) qui suit cette même problématique de séparation des concepts (parcours et sémantique). La programmation par aspects [98] a aussi été introduite pour la séparation des parties fonctionnelles et non-fonctionnelles (applicative ou de services) d'une application. Les approches par transformation de programme utilisées pour la programmation par aspects ont montré leurs limites [32]. Il est clair qu'il existe des liens très forts avec les travaux de recherche sur la réflexivité [112, 113]

```
Cosynt for tiny is

Concrete Syntaxe {
   BNF {
      program(Decl, Stats)  : *[ #Decl ] #Stats ;
      statements(List)      : %LCURLY *[ #List ] %RCURLY ;
      while(Cond, Stats)    : "while" %LPARENT #Cond %RPARENT #Stats ;
      booleanDecl() @varName: "boolean" @varName ;
      intDecl() @varName    : "int" @varName ;
      var()                   : getValue(%VAR) ;
      string()                : getValue(%STRING) ;
      int()                   : getValue(%INT) ;
      equals(Left, Right)   : #Left "==" #Right;
      notEqual(Left, Right) : #Left "!=" #Right;
      plus(Left, Right)     : %LPARENT #Left "+" #Right %RPARENT;
      minus(Left, Right)    : %LPARENT #Left "-" #Right %RPARENT;
      mult(Left, Right)     : %LPARENT #Left "*" #Right %RPARENT;
      div(Left, Right)      : %LPARENT #Left "/" #Right %RPARENT;
      affect(Var, Exp)      : #Var "=" #Exp ";" ;
      true()                  : "true";
      false()                 : "false";
      if(Cond, Then, Else)  : "if" %LPARENT #Cond %RPARENT #Then "else" #Else;
      println(Vals)         : "println" %LPARENT *[ #Vals separator "+" ] %RPARENT ";";
   }

   Parser[k = 2] {
   }

   Lexer[k = 1, attributes = VAR] {
      VAR     = <('a'..'z'|'A'..'Z'|'_'|'$')('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'$')*> ;
      INT     = <('0')|(('1'..'9')('0'..'9')*)> ;
      STRING  = <'&quot;'('a'-'z'|'A'-'Z')*'&quot;'> ;
      RPARENT = <')'>;
      LPARENT = <'('>;
      RCURLY  = <'}'>;
      LCURLY  = <'{'>;

   }
}
```

FIG. 1.8: Exemple complet de description en COSYNT pour notre petit langage TINY

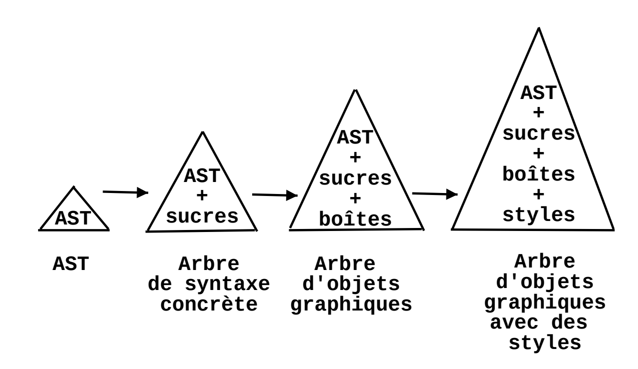


FIG. 1.9: Représentation des transformations de l'AST à l'arbre d'objets graphiques

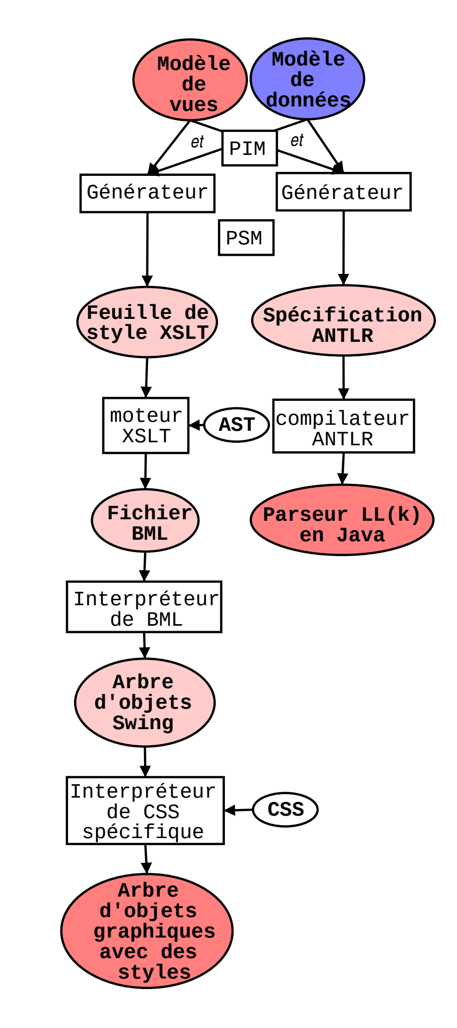


FIG. 1.10: Schéma pour l'implantation de notre langage COSYNT

pour les langages à objets, en particulier la notion de MOP (*Meta-Object Protocol*) [99]. Les mécanismes mis en jeu dans ces approches totalement dynamiques ne sont pas simples

d'utilisation. Ils demandent de comprendre la sémantique sous-jacente des langages à Objets.

L'originalité de notre approche est de partir d'une spécification déclarative de la structure des objets et d'effectuer une génération de code source des visiteurs enrichie par les mécanismes de programmation par aspects ou adaptative. Plus précisément, lors de la génération du code source (les visiteurs), il est facile d'ajouter du code pour gérer (dynamiquement) un branchement d'aspect. L'intérêt de cette approche est, d'une part, d'éviter les problèmes d'efficacité liés à l'utilisation de la réflexivité, et d'autre part, de cacher à l'utilisateur la complexité des mécanismes mis en jeu. De plus, notre approche de programmation par aspects, adaptée à nos besoins, a le mérite d'être d'une mise en œuvre très simple, par une extension naturelle du patron de conception visiteur (nos générateurs). Avec tous ces mécanismes « cachés », une analyse peut, très facilement, être étendue, soit par héritage, soit par l'ajout d'aspects.

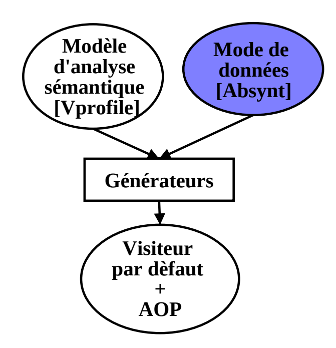


FIG. 1.11: Les générateurs pour la sémantique

ABSYNT et le langage VIPROFILE forment un modèle (PIM) utile pour définir des visiteurs à signatures de méthodes et à parcours configurables, par simple génération de classes Java (PSM). Comme Java effectue une résolution de méthodes sur les types statiques des arguments (et non dynamiques comme Smalltalk) et pour éviter l'emploi de la réflexivité, il est nécessaire de générer du code supplémentaire pour effectuer l'indirection, de manière transparente pour les utilisateurs. Nous profitons de cette phase de génération de code pour enrichir les visiteurs d'une technique de branchement dynamique d'aspects sur les méthodes `visit`.

Contrairement aux autres systèmes par aspects, nous n'utilisons ni la transformation de code ni la réflexivité ; la transformation de programme ne permet pas l'ajout d'aspects à l'exécution (technique statique) et pose des problèmes lors de références aux méthodes de la classe parente (cas de la méthode `super`), et la réflexivité des problèmes d'efficacité et de passage au niveau méta. Ceci est possible car nous proposons une spécialisation de la programmation par aspects uniquement sur les programmes écrits à base du patron de conception visiteur. Lors de la phase de génération des visiteurs par défaut, il est possible de préparer le code source pour accepter le branchement dynamique d'aspects. Nous avons mis en œuvre une solution adaptée à nos besoins.

Les visiteurs écrits par les utilisateurs ont un code simple, indépendant de toute tech-

nique d'implémentation et peuvent être enrichis, à l'exécution, de code supplémentaire sans être modifiés (sans transformation de programme).

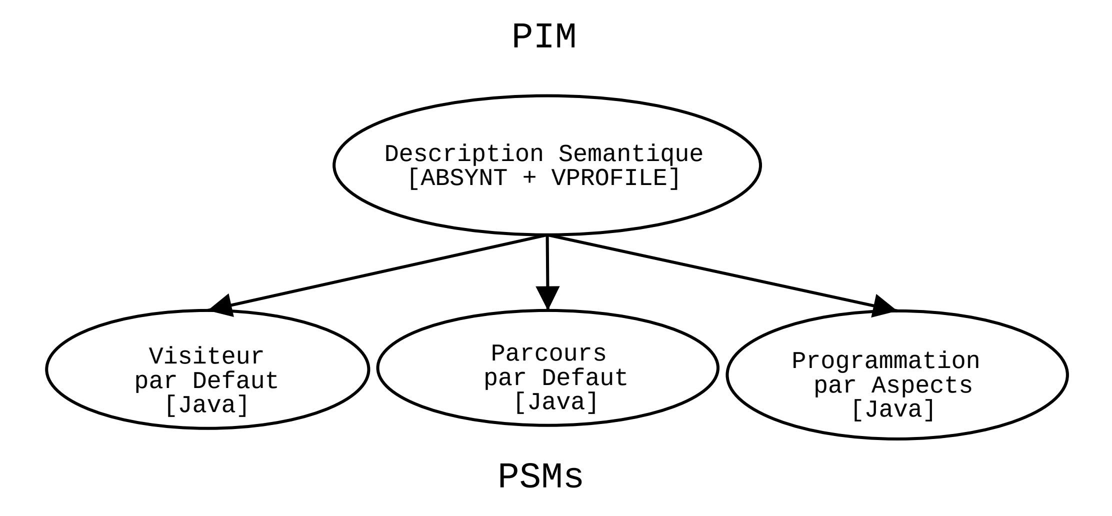


FIG. 1.12: les modèles PIM et PSM pour notre langage VIPROFILE

Finalement, la figure 1.13 résume l'ensemble des outils générés à partir de nos modèles de descriptions pour un langage de programmation.

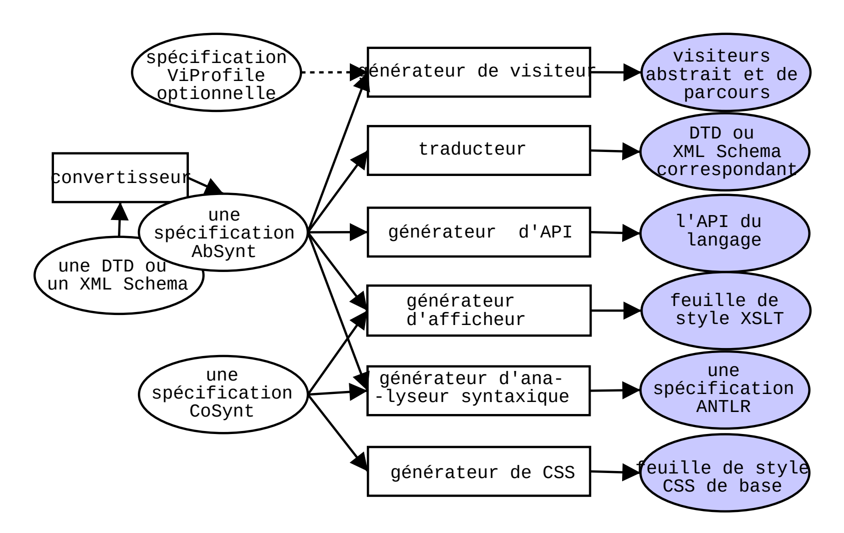


FIG. 1.13: Outils générés à partir des spécifications ABSYNT COSYNT et VIPROFILE

1.2.4 Architecture à composants

Pour l'architecture de notre logiciel, nous avons préféré suivre une approche comparable à MDA en concevant notre propre modèle abstrait de composants (PIM), dédié à nos besoins (métier) et projetable vers des modèles spécifiques concrets (PSM). Les raisons d'un tel

choix sont les suivantes :

- la difficulté de choisir une technologie de composants pérenne et adaptée à notre métier ;
- l'identification claire des besoins en définissant ce modèle indépendamment de toute technologie.

Avec cette approche, décrite plus précisément dans le chapitre 4, les besoins spécifiques sont clairement identifiés. Alors qu'une utilisation directe d'un modèle existant, non adapté à nos besoins, aurait caché les spécificités de nos composants. Lors de la phase de projection de notre modèle vers CCM, EJB ou les *Web Services*, nous nous sommes aussi aperçus que certains de nos besoins, dont l'extensibilité de services des composants, auraient été difficilement exprimables. De plus, grâce à cette technique de projection, il est très facile, par définition de nouvelles règles de transformation, d'exporter nos composants vers une plate-forme ayant un nouveau modèle de composant.

Une architecture par composants pour un tel outil (à génération de code) est nécessaire afin d'établir une nette séparation entre le code du noyau, des outils génériques et des langages. De cette façon, seuls les composants utiles peuvent être chargés en mémoire. Cette modularité est aussi souhaitable pour, aisément, exporter ou importer des composants. Cet aspect de l'architecture à composants de SMARTTOOLS sera développé dans le chapitre 4. Mais il ressort de cette étude que l'approche par composants pour le développement d'une application est fort utile et permet de factoriser (auto-générer) un large sous-ensemble du code grâce aux générateurs. En effet, nous avons pu automatiser grandement la construction de nos environnements (ateliers) et rendre nos applications beaucoup plus homogènes (même discipline de programmation) et exportables.

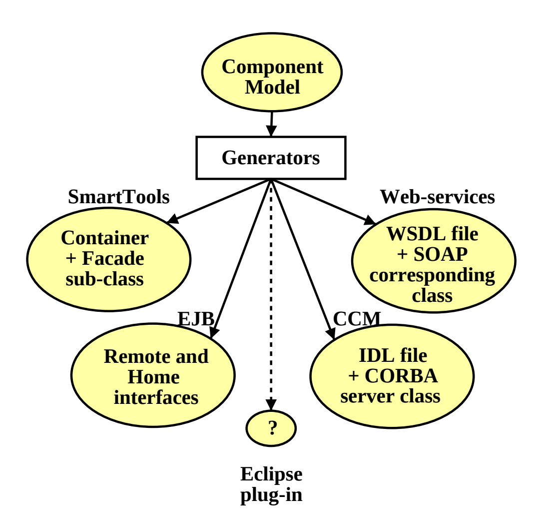


FIG. 1.14: Les transformations de modèles de composants

Nous avons choisi de concevoir notre propre modèle abstrait de composants (PIM), dénommé CDML, répondant à nos besoins spécifiques et de l'implémenter [54]. Afin de rendre nos composants exportables, ce modèle est projetable vers des modèles spécifiques tels que *EJB*, *CCM* ou les *Web-Services* (PSM) [164]. Avec cette caractéristique, il serait

possible, par exemple, de créer des adaptateurs à nos composants pour qu'ils puissent être utilisés dans l'environnement *Eclipse* [9] (un nouveau PSM).

1.3 Les perspectives en terme de champs d'application

Dans cette partie, nous présentons diverses possibilités d'utilisation de SMARTTOOLS dans des domaines très variés. Pour étayer/justifier nos propos, nous indiquons aussi les contrats ou les propositions de collaboration avec des équipes extérieures. Cela montre que nos perspectives d'utilisation sont concrètes et donne une meilleure vision des atouts de l'outil. Cette présentation a un caractère très publicitaire et est organisée en six parties selon les catégories des perspectives de partenariat.

Les langages de programmation ou les langages dédiés

Le champ d'application « naturel » est la conception d'environnements de développement pour des langages de programmation ou langages dédiés. De plus, la conversion DTD (ou XML Schema XML) vers ABSYNT facilite l'introduction de nouveaux langages métiers puisque ces derniers sont de plus en plus définis à l'aide de ces langages de définition de structure. L'exemple type est l'environnement pour l'outil *ant* (un outil équivalent de *make* mais pour des applications écrites en Java) créé, sans réel effort, juste à partir de la DTD de son langage de *makefile*. Nous avons aussi conçu un environnement[6] pour Java Card, un langage de programmation des cartes à puce, ainsi que pour son *byte-code*, dans le cadre d'un contrat industriel avec Bull CP8.

Les langages du W3C

Pour la réalisation des passerelles entre les formalismes du W3C et nos langages de spécification, des environnements de programmation pour les DTDs et les XML Schemas ont été développés. D'autres environnements pour XSLT, SVG, CSS et pour un XML générique ont aussi été créés.

L'environnement XML générique a été réalisé pour permettre de manipuler n'importe quels documents XML, mais avec une granularité moins fine (arbres moins typés) qu'avec un environnement dédié. Ce langage (la syntaxe abstraite) est uniquement composé d'éléments et d'attributs[7].

Tout cela illustre que les formalismes du W3C sont potentiellement des champs d'application pour SMARTTOOLS qui, grâce à son approche générique, rend la création d'environnements de programmation rapide. Pour preuve, SMARTTOOLS a été proposé comme démonstrateur des technologies XML, dans le cadre du projet Européen QUESTION-HOW[8]

[6] Cet environnement n'a pas été porté en version 4.

[7] Toutes les autres informations (*processing*, commentaires, DTD interne, etc) ne sont pas traitées. Pour ce « langage » particulier, il n'y a pas de correspondance entre les noms des balises et les noms des nœuds.

[8] Quality Engineering Solutions via Tools, Information and Outreach for the New Highly-enriched Offerings from W3C : Evolving the Web in Europe http://www.w3.org/2001/qh/

du W3C. Il montre comment une application complexe comme SMARTTOOLS peut être construite en intégrant, de manière homogène et à différents niveaux, les technologies du W3C.

Les langages de description de composants

Les équipes de recherche sur le thème des composants (en particulier les travaux autour de la nouvelle norme CORBA - CCM - ou les plates-formes comme *ObjectWeb*) sont des utilisateurs potentiels de SMARTTOOLS. Des environnements de développement peuvent être générés pour leurs langages métiers (par exemple le langage IDL - *Interface Definition Language*) et des générateurs (compilateurs) réalisés à l'aide de nos techniques.

Les langages de méta-modélisation

Depuis l'émergence d'outils de méta-modélisation autour d'UML (*Unified Modeling Language*), des analogies entre leurs méta-langages, tels que le MOF (*Meta-Object Facility*) et l'OCL (*Object Constraint Language*), et les langages de programmation commencent à être identifiées. Nos techniques de programmation par aspects pourraient être utilisées pour la description de la sémantique de ces modèles (*Action Semantic* d'UML). Dans ce contexte, j'ai participé à plusieurs soumissions d'actions ou projets de recherche.

Une proposition[9] d'ARC (*Action de Recherche Coopérative*) avait été soumise en 2001 , avec les équipes GOAL de Lille, Hector de Toulouse, Triskell de Rennes et du pôle de modélisation de Nantes pour comprendre les points de synergie des différentes familles technologiques telles que les grammaires formelles, les architectures de méta-modélisation, les langages de description de données basés sur XML, et les systèmes d'ingénierie ontologique.

Une proposition de projet RNTL (*Réseau National de recherche et d'innovation en Technologies Logicielles*), de nom XLUC [10] (*XML Languages UML Contrats*), avait aussi été soumise l'année passée, avec les équipes UNSA-CNRS-I3S, UBS/Valoria, SOFTEAM - l'une des entreprises spécialistes d'UML en France - et le crédit agricole breton afin de développer un environnement pour générer, à partir d'un modèle UML, vers plusieurs langages cibles et ayant une extension d'OCL comme langage de contrats.

Un projet RNTL plate-forme, Modathèque a été soumis à l'appel 2003, en collaboration avec SOFTEAM et Thalès en particulier et bien d'autres partenaires [11]. La mise en commun des travaux d'*Objecteering*™, l'AGL de SOFTEAM, et de SMARTTOOLS devrait être le point de départ de futures collaborations.

Les systèmes d'ingénierie ontologique (RDF) du Web sémantique

[9] http://www-sop.inria.fr/oasis/personnel/Didier.Parigot/SAM
[10] voir le résumé à http://www-sop.inria.fr/oasis/personnel/Didier.Parigot/SmartTools/XLUC.html
[11] voir le résumé à http://www-sop.inria.fr/oasis/personnel/Didier.Parigot/MDA/resumer.html

Le thème de recherche du Web sémantique regroupe tout un ensemble de notions très variées dont le lien avec SMARTTOOLS semble être l'analogie entre les systèmes d'ingénierie ontologique [56] (RDFS - *Resource Description Framework Schema*) et les notions de syntaxe abstraite (système de type). Il nous semble raisonnable de penser que nos techniques (visiteur ou programmation par aspects) pourront être utilisées pour certaines applications de ce domaine.

SMARTTOOLS est impliqué dans l'action nationale de recherche de développement SYNTAX (qui vient de débuter), regroupant plusieurs projets INRIA et aussi des industriels dont France Télécom, EADS, Thalès, CEA, EDF et Dassault Aviation. Le but de cette action est de fournir une plate-forme d'intégration et de gestion de documents spécialisés. Dans ce cadre, nous espérons approfondir le rapprochement avec le domaine des ontologies.

Les langages métiers de SMARTTOOLS

L'outil utilise intensivement les technologies objets et XML pour :

– la description de ses langages métiers (définition d'AST avec ABSYNT, de forme concrète avec COSYNT, de profils des méthodes des visiteurs avec VIPROFILE et de l'interface avec LML) et des composants avec CDML,
– ses fichiers de configuration (scripts de lancement),
– la personnalisation des vues graphiques (CSS).

Cette utilisation intensive, à différents niveaux, valide nos choix et cela est susceptible d'engendrer de nouveaux champs d'application. Les outils de SMARTTOOLS présentent la caractéristique d'être directement utilisés pour définir ses propres langages métiers, ainsi nous avons donc du suivre une approche pragmatique dans la conception de ces derniers. Par exemple nos outils de transformation des formalismes DTD ou XML Schema ont été conçus avec une volonté de répondre rapidement à nos besoins.

1.4 Les relations avec mes travaux sur les Grammaires Attribuées

Mes travaux de recherche sur les grammaires attribuées, présentés dans le chapitre 2, ont fortement influencés les diverses approches prises lors de la conception de SMARTTOOLS, cela à plusieurs niveaux. En effet, on retrouve dans SMARTTOOLS de nombreux concepts issus de l'outil FNC-2 [91, 88], un système de traitement de GA que j'ai développé avec Martin Jourdan [85] pendant dix ans après ma thèse [130, 131].

Les aspects essentiellement de l'outil FNC-2 un système de traitement de grammaires attribuées, que l'ont retrouve dans SMARTTOOLS sont principalement les suivants :

1. une notion de syntaxe abstraite étendue (notre langage ASXdans FNC-2) avec ce même rôle de langage pivot ou d'interface que joue notre langage ABSYNT dans SMARTTOOLS, indépendant d'un langage de programmation donné. Par exemple, j'ai dû concevoir diverses passerelles entre ce langage ASX vis-à-vis d'autres formalismes, comme METAL pour CENTAUR [31], FSDL[42] pour le projet COMPARE (cf la section 2.19.1, ou encore les types concrets de CAML [108].

2. un langage de description d'affichage graphique (PPAT) [92] (cf section 2.19.2 et de construction d'arbre abstrait (ATC).
3. une architecture par composants (Frond-end, générateur d'évaluateurs d'attributs, divers back-end vers des langages cibles) avec une sérialisation des informations véhiculées.
4. une notion forte de génération de code. Dans FNC-2 , cela était poussé à l'extrême, avec plusieurs générateurs de code source (C, lisp, CAML).

Par contre, FNC-2 n'offrait pas un environnement interactif (interface utilisateur) comme SMARTTOOLS et cela a été un handicap important pour la diffusion de ce dernier. Cela a motivé mes travaux sur l'utilisation de CENTAUR pour construire cette interface utilisateur pour notre outil FNC-2 [23].

Sur les aspects traitements sémantiques, le code produit par FNC-2(évaluateur d'attributs) peut s'apparenter à la technique des visiteurs. Les évaluateurs d'attributs sont composés de « séquences de visites » qui peuvent être comparées à des visiteurs.

Mais l'aspect le plus intéressant me semble être le lien que l'on peut établir entre la programmation par aspects et la programmation par attributs. En effet, la programmation par attributs a comme qualité première de permettre une programmation incrémentale, par préoccupation, d'une application. Lors du développement du système FNC-2, j'ai utilisé fortement cette qualité pour la définition de notre langage OLGA. Avec ce style de programmation par attributs, il est possible d'ajouter facilement, par la définition d'un nouvel attribut, une nouvelle préoccupation, à une application sans défaire complètement l'application. Je reviendrai sur ce point dans le chapitre 2. De plus, nos travaux sur le couplage de grammaire [107] peuvent être comparés à la transformation de modèle (l'approche MDA). On cherche une grammaire minimale (un PIM) pour une certaine préoccupation que l'on va pouvoir projeter sur différents langages (des PSMs).

1.5 Les perspectives de recherche

L'intérêt de ces travaux, est essentiellement de concrétiser, à travers la réalisation de SMARTTOOLS, l'approche par programmation générative pour une famille d'applications. Plus précisément, cela nous permet de mieux faire comprendre la notion de fabrique d'application qui est l'idée forte de cette approche. En effet, cette notion de fabrique est l'une des réponse aux nouveaux défis du génie logiciel (qui depuis quelques années subit d'importants bouleversements) pour produire des applications ouvertes et évolutives. Cette notion de fabrique (d'usine) semble être la plus prometteuse et elle répond au mieux par ses caractéristiques, à ces nouvelles contraintes pour le développement de logiciels.

Fabrique ou usine de production d'application

En effet, pour assurer une meilleure évolution et sûreté dans les développements de logiciel dans le contexte de l'informatique ubiquitaire, il est essentiel qu'une grande souspartie des applications soit produite sous la responsabilité de générateurs de code source. Ces générateurs à partir de modèles ou de descriptions abstraites produisent et contrôlent les

parties de l'application qui suivent un même schéma de conception (patron de conception) ou/et dépendent fortement d'aspects très techniques liés à des préoccupations orthogonales à l'application de base (comme la distribution, ou la sécurité). Ainsi le développeur peut se concentrer exclusivement sur les parties métiers de son application, la valeur ajoutée, et laisser la responsabilité à la fabrique de produire l'empaquetage de ces dernières.

Nous rentrons dans l'ère de la conception de logiciels à l'aide de fabriques (d'usines) qui prennent en charge les parties à fort potentiel d'évolution des applications. Ces fabriques prennent aussi en compte les assemblages de plus en plus complexes pour concevoir les systèmes informatiques de demain. Je pense que SMARTTOOLS est un bel exemple (un démonstrateur) de cette nouvelle démarche pour le **génie logiciel** des prochaines années.

Les composants

Le succès de la notion de composant vient certainement de cette nouvelle préoccupation. Par comparaison, les modèles de composants, en particulier ceux proposées par les industriels (EJB ou CORBA), correspondent à des fabriques pour les applications distribuées. Ces fabriques offrent gratuitement les divers services pour produire rapidement une application distribuée. La notion de composant permet d'encapsuler le savoir-faire utile pour construire des applications distribuées et permet au développeur d'utiliser sans effort cette compétence. Cette utilisation peut difficilement se réaliser d'une manière classique, par héritage ou délégation.

Les patrons de conception

De la même manière, le succès des patrons de conception est issu du même besoin. Les patrons de conception définissent d'une manière abstraite ou générique des méthodologies de conception. Les problèmes rencontrés avec cette approche sont de natures suivantes :

- la concrétisation du concept est complètement à la charge du programmeur et cela peut entraîner des erreurs d'interprétation et d'implantation. Il existe peu d'outils qui produisent directement le code source à partir d'un modèle.
- l'utilisation est souvent une variante ou une combinaison d'un ou plusieurs patrons à la fois.
- l'identification d'un site d'utilisation d'un patron s'effectue souvent trop tard dans le développement d'une application.

L'idée sous-jacente d'une fabrique est bien d'identifier, à l'avance, les divers patrons de conception qui vont être nécessaires pour une famille d'applications. Les générateurs vont captés cet ensemble de patrons de conception et leurs variantes utiles pour ce type d'application. Mais surtout, ils vont faciliter l'évolution de la mise en œuvre sur l'ensemble des applications produites puisque la concrétisation est sous la responsabilité de la fabrique, des générateurs.

La méta-modélisation

De même, les approches à base de modélisation UML essaient elles-aussi de répondre à ce besoin. Ces approches cherchent à modéliser les diverses expertises métiers de conception autrement qu'à travers la programmation, l'implantation. Pour les industriels, l'important est de pouvoir avoir un moyen efficace de spécifier et de modéliser leurs savoir-faire pour une meilleure réutilisation et mémorisation de leurs expertises métiers.

Il est clairement admis, maintenant, que les langages de programmation n'ont finalement pas cette vocation ou remplissent très mal ce rôle de mémorisation d'un savoir-faire. Ils restent encore au niveau de l'algorithmique d'une application (bibliothèque), un niveau assez bas dans la conception d'une application. De plus, depuis quelque temps, l'approche UML par l'introduction de la notion de méta-langage, MOF, rejoint d'une certaine manière notre approche. En effet, l'approche par modélisation s'oriente vers la définition de divers modèles spécifiques par opposition au modèle UML qui essaie de définir un seul modèle unifié pour modéliser les langages de programmation à objets.

Les différents niveaux hiérarchiques de modèles

Mais nos travaux montrent aussi que ce concept de fabrique d'applications peut, d'une part, s'appliquer à différents niveaux au sein même des entités qui composent une application (même sur les parties métiers) et, d'autre part, qu'elle doit être accessible, utilisable par le développeur, à travers des modèles indépendants de toute technologie et langage de programmation.

En effet, nous montrons que les phases de génération s'appliquent à divers niveaux que l'on peut hiérarchiser de la manière suivante : Ceux qui intervient au sein même du composant sur les aspects de définition de structure de donnée, aide à la programmation des traitements spécifiques sur les structures de données sous-jacent et enfin sur les aspects de représentation graphique de la forme logique (structure de données). Ceux qui définissent l'interface du composant (le modèle de composant) et enfin ceux qui définissent l'architecture de l'application (création des instances, déploiement, etc).

Chaque niveau manipule des concepts différents et produit des entités de granularité différente. La cohérence d'une fabrique dépend fortement de la communication entre les divers modèles de chaque niveau. Dans notre cadre, nous montrons l'intérêt de traiter, en même temps, ces divers niveaux. Par exemple, la présence du modèle de données de nos composants nous permet d'effectuer des services complexes entre nos composants (sélection d'un noeud) qui dépendent fortement du contenu interne des composants, sans pour autant nuire à la séparation fonctionnelle de ces derniers. L'utilisation d'un format neutre (XML, dans notre cas) permet cette indépendance entre le modèle et les diverses implantations sous-jacentes aux différents types de composant. En effet, nos composants logiques (arbre de syntaxe abstraite) implémentent le modèle d'une manière très différente de nos composants graphiques sans pour autant interdire un quelconque échange d'information complexe entre eux.

Nous défendons donc une approche d'architecture logicielle, à boite grise (en opposition à boite noire) pour les différents niveaux de granularité des entités qui composent une application. Mais nous demandons que chaque abstraction (boite grise) soit tel-que la perte d'information (abstraction) correspond bien aux besoins du niveau hiérarchique considéré.

Modèles dédiés par opposition à modèles génériques

De plus, nous somme plus convaincus de l'intérêt d'une approche par famille d'applications (une fabrique pour une famille d'applications) qu'à une approche basée sur la notion d'un modèle générique (par exemple, les technologies par composants EJB, ou encore l'approche UML pour les langages de programmation par objets). En effet, il nous semble préférable de définir un modèle adapté au vocabulaire du métier sous-jacent. Ce modèle caractérise parfaitement l'expertise métier sous-jacent à la famille d'applications considérées, sans obscurcir le modèle par des considérations techniques d'implantation. Notre modèle de composants pour SMARTTOOLS est un exemple de cette approche. Cela explique et motive la proposition de l'approche MDA de l'OMG qui cherche elle aussi, à s'abstraire des aspects spécifiques à une technologie d'implantation pour mieux identifier les notions associées à une famille d'application, à un métier particulier.

Finalement, nous croyons plus à des modèles abstraits (indépendants d'une technologie) dédiés à un domaine (application distribuée, application langage) qu'à un modèle générique (universel). On peut faire le rapprochement avec les langages de programmation (universel) qui se heurtent très souvent à cette difficulté d'évolution de l'informatique et de pénétration du marché. La notion de langage de programmation se vend toujours très mal de nos jours si ce dernier n'apporte pas une réelle avancée facilement mesurable pour le programmeur. L'approche traditionnelle issue de ce domaine de conception des langages de programmation ne peut plus s'appliquer comme telle dans ce nouveau contexte de développement logiciel qui évolue extrêmement rapidement.

Mais cela est encore plus vrai, face à la vitesse de création de nouveaux domaines d'applications, de technologies (matériels) etc. Il semble donc qu'avec l'apparition de l'informatique dite ubiquitaire, les logiciels aient besoin de s'adapter très vite à ces nouvelles technologies qui apparaissent chaque jour. Ils doivent les utiliser et les prendre en compte au cours de leur cycle de vie.

Meilleure maintenance et évolutivité

Tout ceci, nous fait penser qu'il est vital pour la fabrique (les générateurs) et les applications générées qu'elles soient elles-mêmes adaptables pour pouvoir prendre en compte ces besoins d'ouverture et d'évolution. C'est ici que l'on trouve l'un des grands intérêts de cette approche par programmation générative. Les générateurs contrôlent et génèrent les parties effectivement sensibles à ces évolutions (communication avec l'environnement d'exécution). Ainsi cette main mise des générateurs sur une large partie du code source d'une application, facilite grandement la maintenance de cette dernière. En effet, il suffit de modifier les générateurs pour prendre en compte ces évolutions.

Séparation des préoccupations

De plus, si la conception de ces générateurs reste ouverte, cette démarche permet de prendre en considération des besoins très spécifiques au domaine sans pour autant demander une redéfinition complet du modèle. L'intérêt de l'approche par séparation des préoccupations vient de cette idée qu'il est important de pouvoir développer une application de

manière incrémentale. Cette thématique par programmation par aspects a donné les moyens d'une telle approche et répondait à un besoin récurent. Plus précisément, cela permet de répondre en même temps aux constatations suivantes :

- on a besoin d'intégrer dans une application des fonctionnalités non prévues initialement ;
- les retours fréquents entre la phase d'implantation et la phase de spécification sont de plus en plus monnaie courante. Il ne reste plus beaucoup des développements d'application où les schémas classiques de conception s'appliquent correctement. Un développement dit incrémental est devenu un besoin récurrent pour beaucoup de domaines.
- la maintenance des applications est devenue la partie la plus longue (et donc coûteuse) dans le cycle d'un logiciel. Des outils ou de nouvelles méthodes de programmation doivent voir le jour pour répondre à ces nouveaux défis.
- les successives adaptations (ou modifications) transforment et rendent l'application de moins en moins lisible. Avoir un moyen d'adapter une application sans obscurcir le code initial a donc vu le jour.
- la complexité des applications et de leurs environnements d'exécution a introduit diverses préoccupations que l'on voudrait mieux modulariser. En effet, ces divers aspects sont souvent intégrés à de multiples endroits dans l'application de base et cela rend la maintenance de l'application très difficile.

Tout cela explique le succès de cette nouvelle thématique qu'est la programmation par séparation des préoccupations, dont la programmation par aspect est l'une des solutions. Nous pensons que cette approche peut être utilisée pour rendre les divers générateurs adaptables aux spécificités d'un domaine par l'ajout d'aspect spécifiques aux besoins. Ici, nous essayons d'appliquer cette approche à la programmation générative. Il nous semble important de laisser ouverts les générateurs et de permettre une réutilisation des concepts sous-jacents pour des modèles différents. Il est donc important de pourvoir réutiliser les aspects sur d'autres modèles suivant une transformation aux niveaux des modèles.

Transformation de modèle

L'idée de base de cette notion est la transformation structurelle de programmes, permettant de projeter automatiquement une sémantique d'un modèle vers un autre. C'est une notion que j'ai étudiée, en particulier pour les programmes écrits en grammaire attribuée (voir les sections 2.13 et 2.15).

Notre intérêt pour la programmation par aspects découle de ce rapprochement. En effet, une autre propriété de la programmation par aspects est de permettre de mieux décomposer un programme ;les structures de données, les parcours récursifs sur celle-ci (flux de contrôle) et les actions proprement dites (code des aspects). Cette propriété permet une manipulation (transformation) plus aisée des programmes par projection des aspects conformes à une transformation sur la structure de données (voir la figure 1.15). En effet, dans le contexte de la nouvelle approche MDA, les transformations de modèle sont la pierre de voûte pour le passage entre les PIMs (*Platform Independent Models*) et les PSMs (*Platform Specific Models*). Par exemple (voir la figure 1.16, s'il existe une transformation entre notre

modèle de composant et le modèle pour CORBA (IDL), alors il est possible d'obtenir gratuitement un générateur de conteneurs (pour notre architecture) avec comme entrée les spécifications en format IDL. Pour cela, nous pensons fortement nous inspirer de nos travaux de recherche [47]) sur le concept de déforestation [25, 165, 150], ou plus précisément sur la composition structurelle [67, 76] introduite dans la théorie des Grammaires Attribuées (voir les sections 2.13 et 2.15). De plus nos travaux sur le couplage de grammaire [107] (construction semi-automatique des transformations de modèle) pourra certainement trouver un nouveau champ d'application. Nous pouvons aussi citer les travaux sur les bases de données semi-structurées [121] qui cherchent à définir des transformations (automatiques) sur les Schemas XML [14, 29].

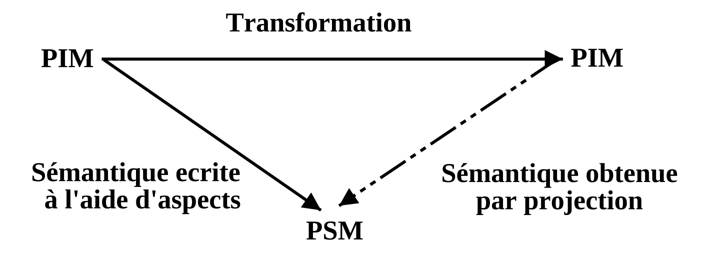


FIG. 1.15: Transformation de la sématique d'un modèle

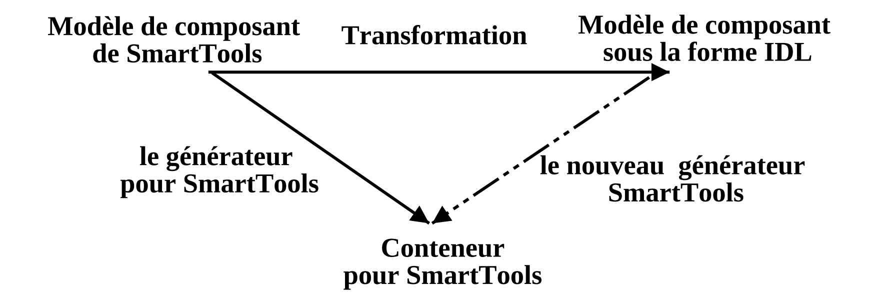


FIG. 1.16: Exemple de projection d'une sémantique

Notion de composants méthodologiques

Notre démarche caractérise aussi la notion de composant méthodologique issue de l'approche MDE (*Model Driven Engineering*). En effet, notre outil fournit un ensemble de composants méthodologiques pour la conception d'interfaces utilisateurs, de traitements sémantiques, ou de projections de modèle de composant. Nos générateurs sont des composants MDE.

Plus précisément, le composant “modèle de composant” prend en entrée une instance du modèle (PIM) et le transforme vers les divers PSM. Le composant de traitements sémantiques prend en entrée le modèle (PIM) et le transforme (produit) en un ensemble de classe Objets. Dans c'est deux cas, les transformations sont figées par l'implantation du générateur.

L'un des plus complets est certainement le composant d'interfaces graphiques (COSYNT). Il prend en entrée le modèle de données (PIM en ABSYNT) et les transformations du PIM vers les PSMs décrites dans un langage (langage COSYNT) de transformation de modèles spécifique au domaine sous-jacent. Ainsi ce composant prend en entrée un modèles et des transformations vers des PSMs. Ce composant méthodologie a le mérite d'instancier cette approche sur un exemple simple et montre, là aussi, l'intérêt de définir un langage de transformation spécifique au domaine du composant méthodologique.

Actuellement, l'OMG a lancé un appel d'offre pour la définition d'un langage de transformation de modèle (*Queries, Views & Transformations - QVT*) universel [15]. Nous pensons, comme pour notre notion de fabrique, qu'il est plus judicieux de proposer des langages de transformation pour chaque famille de composants méthodologiques que d'essayer de définir un langage de transformation généraliste.

Les nouveaux paradigmes de programmation

Finalement, nous sommes convaincus que la sûreté et la fiabilité d'une application viendront plus par une production automatisée (donc éprouvée) que par d'autres moyens (comme des analyses statiques ou des techniques de preuve ou encore des modèles théoriques). De plus, ces approches se heurtent à la forte évolution des technologies et donc à difficulté de concevoir des modèles théoriques pertinents et pérennes. Mais surtout, les caractéristiques hautement dynamiques et hétéroclites des applications de demain (communication sans fils, modification des environnements et multi-supports) ne permettent pas de concevoir des modèles formels suffisamment pertinents et précis. Enfin, il nous semble plus important de réfléchir aux nouveaux paradigmes (composants, patrons de conception, programmation par aspects, fabrique ou usine de développement) qui fournissent des outils pour rendre les applications beaucoup plus évolutives et ouvertes. Ces nouveaux concepts recherchent tous à rendre plus ouverts et adaptables les langages de programmation classiques. Le choix d'une approche pragmatique, qui appuie sur les langages de programmation existants (Java, dans notre cas), en proposant que des extensions ou des enrichissements à ces derniers, nous parait être plus prometteur que la recherche de modèles théoriques pour résoudre cet ensemble de contraintes. La forte évolution et les origines diverses ne permettent pas de concevoir un modèle figé et demande une constante adaptation et remise en cause des concepts introduits. Nous espérons que notre outil SMARTTOOLS permet de mieux appréhender cette nouvelle approche du génie logiciel pour les prochaines années.

Deuxième partie

Travaux sur les Grammaires Attribuées

Chapitre 2

Travaux de recherche sur les Grammaires Attribuées

Avant propos

Dans cette partie, je vais résumer l'ensemble de mes travaux sur les grammaires attribuées. Pour chacun des travaux de recherche, j'essaierai d'expliquer la problématique traitée sous un format compréhensible par un non-expert du domaine. Dans ces présentations, j'ai

préféré insister plus sur les solutions mises en œuvre que sur les moyens techniques sous-jacents. Le lecteur pourra se référer aux articles associés (pour la partie théorique) ou au manuel d'utilisateur[1] de notre système de traitement de grammaires attribuées FNC-2 [88], pour se faire une idée de l'ampleur de ces travaux. De même, il pourra lire l'habilitation à diriger des recherches de MARTIN JOURDAN qui présente une sous-parties de mes travaux de recherches [85][2]. En effet, nous avons travaillé plus de 10 ans ensemble sur ce domaine de recherche, les grammaires attribuées. Nous avons maintenu pendant plusieurs années une bibliographie complète sur ce thème et elle compte plus de 1000 références bibliographiques [134][3]. Il est important de noter que les GAs ont été utilisées dans divers domaines de recherche, voir en particulier l'article [124]. Cela montre que ce thème de recherche a été très productif pendant les années 80-90 et est, depuis les années 90, en perte de vitesse (beaucoup moins de publications).

2.1 Introduction

Les grammaires attribuées (GA) [103, 60, 21] est une extension des grammaires non contextuelles[4] permettant de traiter les aspects sémantiques des langages en plus de leurs aspects syntaxiques. C'est une méthode déjà ancienne (une vingtaine d'années) mais qui n'a pas encore rencontré tout le succès que ses qualités, en tant que méthode de spécification, pouvaient laisser prévoir. La raison la plus importante en est que ses implantations — on peut, théoriquement, construire automatiquement à partir d'une GA un programme qui réalise la fonction qu'elle spécifie — ont jusqu'ici été handicapées par un compromis plutôt décevant, entre le pouvoir d'expression et l'efficacité. Mais surtout, il manquait un support de diffusion pour le grand public (hors de la communauté des grammaires attribuées), comme actuellement les technologies XML jouent ce rôle de diffusion, ou encore un support théorique admis par tous comme pour la communauté fonctionnelle avec le lambda calcul.

J'ai volontairement pris le choix d'une présentation très informelle de mes travaux de recherche, pour les deux raisons suivantes. D'une part, mes travaux portent sur un large sous-ensemble des problèmes rencontrés en grammaires attribuées, et dont la formalisation demanderait une trop longue présentation. Cela demanderait de rentrer dans les détails de la théorie des Grammaires Attribuées qui est assez conséquente. La deuxième raison, qui est d'une certaine manière une conséquence de ce qui précède, est que le nombre impressionnant de résultats [5] a souvent rebuté les utilisateurs potentiels des GAs. En effet, l'un des problèmes majeurs dans ce domaine est l'efficacité du programme qui exécute la spécification d'entrée (les évaluateurs d'attributs). La théorie des GAs donne tout un ensemble de résultats théoriques sur ce problème, mais sans pour autant fournir une solution "idéale". Ainsi, lors de l'application de ces résultats, c'est-à-dire de la réalisation du construc-

[1] ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/f2manual.ps.gz
[2] ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/theses/habilit-martin.ps.gz
[3] http://www-rocq.inria.fr/oscar/www/fnc2/AGabstract.html
[4] la notion de grammaire abstraite n'existait pas à l'époque, dans les années 70
[5] voir le nombre de référence, plus de 1000 références, de la bibliographie complète sur les GAs [134]

teur (générateur d'évaluateur d'attributs) qui engendre le dit programme exécutable, il s'est avéré assez difficile de trouver un bon compromis entre les différentes solutions. En effet, à chaque type de constructeur, le compromis interne à ce dernier correspond souvent pour les utilisateurs, à la compréhension d'un des problèmes qui n'est pas forcément d'un abord facile. Par exemple, il n'est pas facile de comprendre pourquoi une GA n'est pas telle classe de GA (la classe acceptée par le générateur). Ainsi les utilisateurs ont été plutôt sensibilisés aux problèmes internes de la construction qu'aux qualités intrinsèques des GAs, sans pour autant se voir proposer une solution satisfaisante. Par exemple, la qualité de programmation incrémentale (pouvoir exprimer à l'aide des attributs, séparément chaque préoccupation qui compose une application) n'a pas, de mon point de vue, été assez mise en avant. Avant l'apparition de la programmation par aspects, il n'existait pas, autre que la programmation par attributs, un langage de programmation par (pour) séparation des préoccupations.

Notre action au sein du projet "Langages et Traducteurs" à Rocquencourt visait à améliorer suffisamment ce compromis pour que les GAs soient enfin reconnues comme un outil important. Cette action se mena sur plusieurs fronts — développement d'applications, formation, méthodologie de programmation par attributs — dont la concrétisation est la réalisation d'un système de traitement de GA, performant, que nous avons nommé FNC-2 .

2.2 Rappel sur la notion de grammaire attribuée

Rappelons brièvement le principe de la méthode des grammaires attribuées. On considère une grammaire indépendante du contexte. À chacun de ses non-terminaux, on attache deux ensembles de symboles : les attributs synthétisés, qui véhiculent de l'information depuis les feuilles d'un arbre de dérivation jusqu'à la racine, et les attributs hérités, qui transportent de l'information en sens inverse. À chaque production, on associe un certain nombre de règles sémantiques qui spécifient comment calculer les attributs de sortie, c'est-à-dire les attributs synthétisés du non-terminal membre gauche de la production, et les attributs hérités des non-terminaux du membre droit de la production, en fonction des attributs d'entrée de la production (les autres attributs). D'après ce rapide exposé, les principales qualités de cette méthode sont les suivantes :

- Les grammaires attribuées sont une méthode de description déclarative, c'est-à-dire non-procédurale : il n'est pas spécifié comment (dans quel ordre) on calcule les attributs, mais seulement ce que l'on calcule (leur valeur).
- Elles sont une méthode structurée, induite par les productions de la grammaire ;
- Elles sont basées sur le principe de localité : tout se passe au sein de chaque production ;
- Elles sont une spécification exécutable : il est possible de construire un programme, appelé évaluateur d'attributs, qui exécute la spécification d'entrée. Un évaluateur est chargé de calculer, à l'aide des règles sémantiques, un ou plusieurs attributs d'un arbre de dérivation qui représentent la valeur sémantiques du texte source.
- Enfin, et cela n'est apparu que relativement récemment, c'est une méthode de développement *incrémentale*, [91] c'est-à-dire qu'on peut développer une grammaire attribuée petit-àpetit, et tester chaque partie de spécification indépendamment des

autres ; je crois que c'est un très grand avantage par rapport à d'autres méthodes et correspond à une programmation par aspects particulière.

La finalité de mes travaux a été d'essayer de remédier aux défauts des GAs (énoncés ci-dessus). En effet, nous sommes convaincus que les GAs, par leurs qualités propres, en particulier la facilité d'écriture, méritent une plus large diffusion. Ainsi nous nous sommes placés dans le cadre de l'utilisation pratique (comme pour langage de programmation), sur des exemples réalistes, et non plus dans un cadre abstrait (théorique). Sous cette hypothèse, nous montrons, à travers la réalisation d'un nouveau constructeur, qu'il existe une solution satisfaisante à ce défaut. Ainsi les utilisateurs potentiels pourront enfin apprécier et juger des qualités de la programmation en GA (par attributs), sans pour cela être obligés de supporter toute la complexité de la construction, et parfois de se plier à ces contraintes éventuelles (classes de GA acceptées par le constructeur). Pour construire cette solution, nous avons étudié un vaste sous-ensemble de problèmes rencontrés dans la théorie des GAs, réexaminé les résultats en se plaçant résolument dans notre cadre, et proposé des approches différentes ou des améliorations significatives. L'ensemble de ces travaux prend encore plus de valeur dans leur juxtaposition. Pour nous, la réalisation de ce constructeur et son utilisation étaient donc un début de "preuve" de la validité de notre approche.

Les deux problèmes importants que l'on rencontre dans la génération des évaluateurs sont les suivants : les évaluateurs d'attributs engendrés sont en général très gourmands en place mémoire, et le temps de construction de ces derniers peut être très important. Il faut bien admettre que ces deux points ont souvent été un frein important à l'utilisation des GAs, surtout sur des données de taille importante. En effet, dans la théorie des grammaires attribuées, les résultats théoriques, en terme de complexité, sont souvent exprimés dans le pire des cas. Mes travaux ont montré que ces résultats d'explosion combinatoire restent confinés à des cas pathologiques de GA, et qu'en pratique sur des textes importants, ces deux points noirs n'apparaissent pas, ou restent dans des limites très raisonnables et en tout cas très loin des bornes définies par la théorie.

2.3 Le système FNC-2 : présentation générale

Nous voulions intégrer dans FNC-2 les développements les plus récents dans ce domaine sur le plan théorique (voir ci-dessus), mais aussi en faire un outil de qualité, c'est-à-dire qui soit à la fois puissant, efficace et d'utilisation aisée et souple. Au début 1990, un premier prototype a été disponible et pouvait être raisonnablement utilisé : la preuve en est qu'il est "bootstrappé" à partir de ce premier prototype. D'après les premiers résultats d'utilisation (cf. la section 2.19.2 comme une première vraie utilisation), nous étions encore plus convaincus que notre système représentait certainement l'une des meilleures solutions (ou compromis) à notre problème de départ. Énonçons donc les qualités principales de ce système FNC-2.

- Grande puissance d'expression : FNC-2 accepte la plus large classe de grammaires attribuées possible qui ne mette pas en cause l'efficacité du code produit,
- Efficacité en temps et en place, aussi bien à la construction qu'à l'exécution des évaluateurs d'attributs,
- Facilité et sûreté d'emploi, mesurées à l'effort nécessaire pour l'écriture et la mise au point d'une grammaire attribuée,
- Et enfin, la souplesse d'utilisation.

Au cours du développement de FNC-2, écriture de grammaire attribuée de taille importante, je me suis rendu compte que la programmation par grammaire attribuée est une forme particulier de programmation par objets (par attributs) grâce à ses qualités :

- de programmation structurée (locale à chaque constructeur) ;
- et de séparation entre le contrôle (le filtrage sur la structure) et la définition du calcul (les règles sémantiques).

Dans [91], nous avons défendu cette idée de développement *incrémental*, par touches successives (attributs ou aspects) qui permet de faire évoluer facilement une application spécifiée en grammaire attribuée.

2.3.1 Présentation du langage d'entrée OLGA

La définition du langage OLGA (Ouf ! un Langage pour les Grammaires Attribuées) a comme principaux objectifs la facilité, la sûreté d'utilisation et la souplesse d'emploi. Cela est dû à la conjonction des deux concepts suivants :

- Le langage d'écriture de GA (la forme externe) est un langage spécialisé conçu pour cet unique objet, et non plus un langage existant, simplement augmenté de notations supplémentaires. Ce langage, nommé OLGA [91, 90, 85] est un langage applicatif et fortement typé, deux conditions indispensables pour offrir à l'auteur de grammaires attribuées une très grande sécurité dans sa programmation. De plus, OLGA inclut des concepts modernes, comme le traitement d'exception, la modularité, le filtrage de modèles, l'inférence de type et le polymorphisme[6]. De plus, la base syntaxique des GAs est une syntaxe abstraite attribuée, permettant à l'auteur de se libérer des contraintes souvent excessives de la syntaxe concrète. Cette approche a été largement

[6]Cet aspect d'OLGA a été faiblement poursuivi au cours du développement de FNC-2

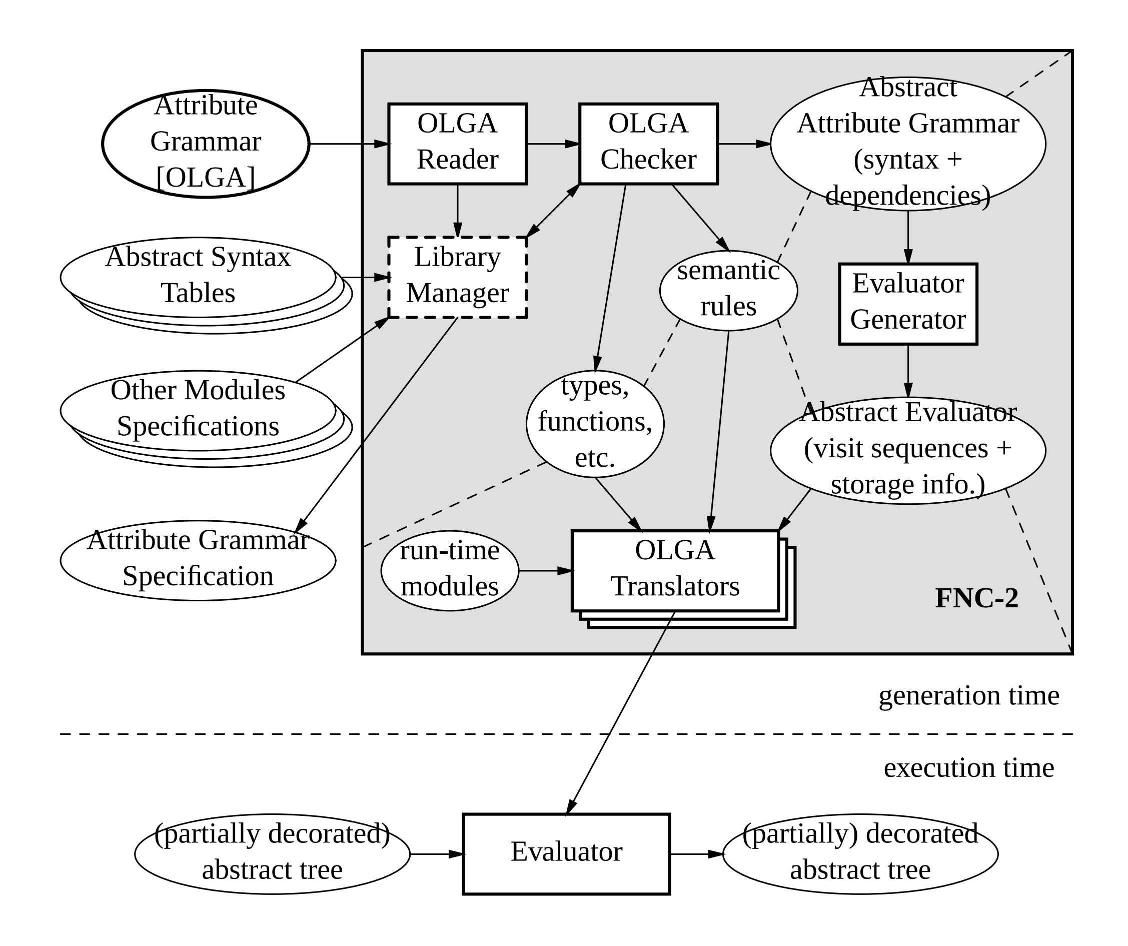


FIG. 2.1: Schéma général de FNC-2

inspirée des travaux de l'équipe CROAP sur le même sujet : le langage METAL. Pour ce faire, on a défini les deux langages suivants : ASX pour décrire une syntaxe abstraite attribuée (déclaration d'un type d'arbre), et ATC pour définir la construction d'un tel arbre à l'aide d'un analyseur lexico-syntaxique classique (comme le système SYNTAX de PIERRE BOULLIER). OLGA est certainement un langage très ambitieux dans ses concepts de base, mais il nous fallait un langage qui se suffise à lui-même, aussi complet que possible. En d'autres termes, ce langage ne doit en aucun cas être un langage "jouet" permettant uniquement de valider les méthodes d'évaluation introduites par le générateur d'évaluateur (cf. la section 2.4) mais un langage permettant de développer des applications en réelle grandeur.

– La souplesse d'emploi vient de l'indépendance totale de la spécification par rapport à son implantation, ce qui se traduit par plusieurs propriétés pour le code produit, les évaluateurs. Ils peuvent être traduits dans plusieurs langages de programmation. Notre première cible a été le langage C puisque le générateur est écrit dans ce même langage, mais d'autres traducteurs ont vu le jour au cours des développements ultérieurs. Ils peuvent être produits soit en version exhaustive, soit en version incrémen-

tale (voir la section 2.8). Ils peuvent être couplés à divers environnements, le langage ASX jouant le rôle de formalisme abstrait (indépendant d'un langage de programmation particulier).

La figure 2.2 montre pour un petit langage Simproc, l'ensemble des fichiers de spécification utile pour réaliser un compilateur de ce langage. On pourra lire en annexe de [88], l'ensemble des spécifications pour ce petit langage SIMPROC.

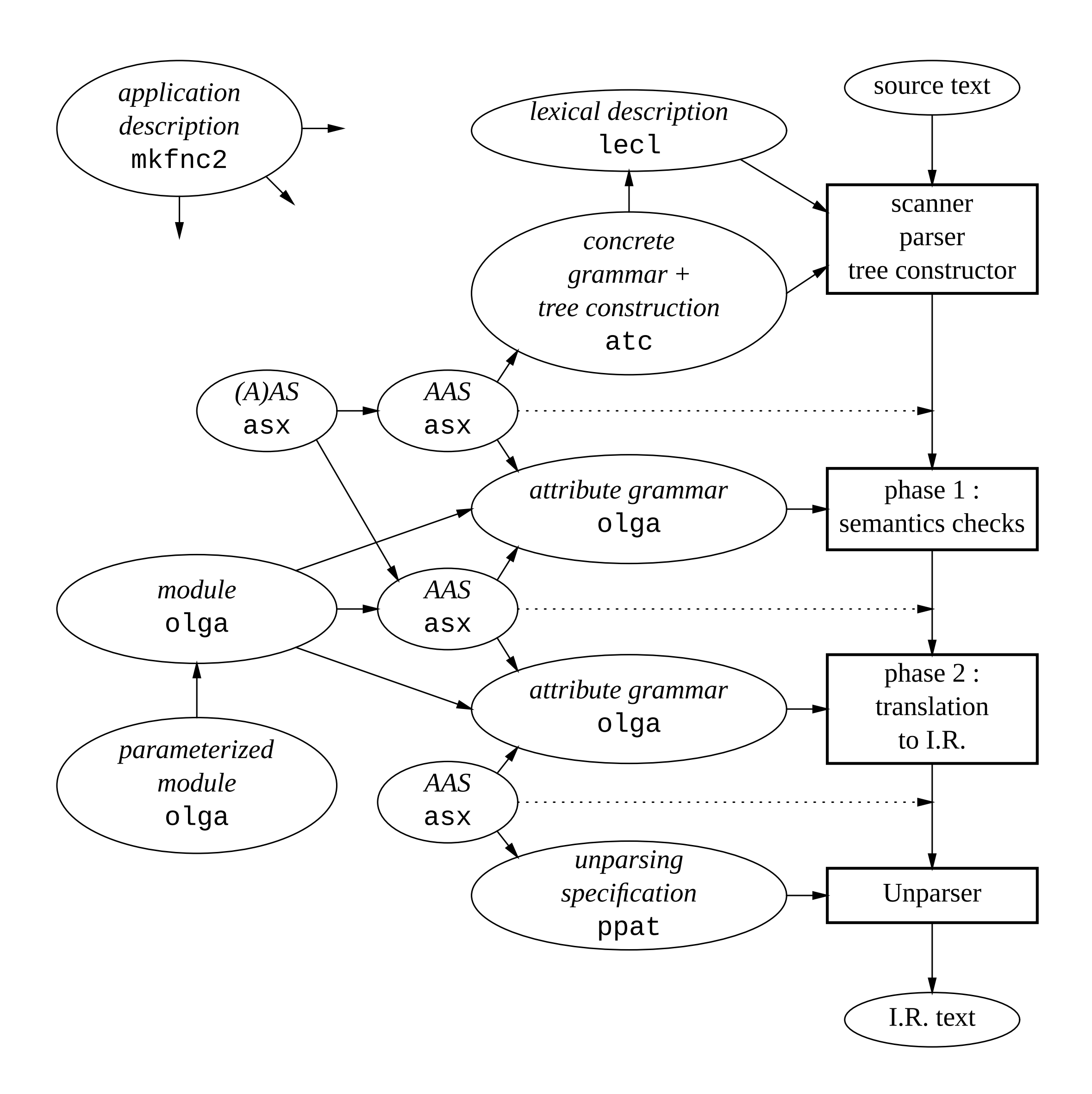


FIG. 2.2: Schéma d'une application de FNC-2

L'ensemble de ces spécifications nous (moi et Martin Jourdan) a amenés à lancer un projet ambitieux. La conception et la réalisation de ce système ne peuvent donc être considéré que comme le travail d'une équipe, composée de MARTIN JOURDAN et de moi-même

(pour les permanents), sans oublier les **(27)** différents stagiaires et **(7)** doctorants (voir les listes des stages et thèses dans le chapitre 5) qui ont participé à des points particuliers du projet. À cause du caractère ambitieux de ce système, nous avons adopté au cours de sa réalisation, une démarche de construction par étapes successives. Ainsi, le système est le résultat d'une superposition de processus à finalités différentes. Il se compose essentiellement des deux parties suivantes : le générateur d'évaluateur, qui réalise la partie purement grammaire attribuée et, autour de ce dernier, un ensemble de modules (la carrosserie [88]) qui réalisent les interfaces vis-à-vis des utilisateurs.

Pendant ma thèse (1986-1988), ma contribution a porté essentiellement sur le cœur du système — le générateur d'évaluateur —, principalement sur la mise en œuvre des nouvelles méthodes d'évaluation d'attributs, dans le but de satisfaire nos deux exigences — c'est-à-dire une grande puissance d'expression et une efficacité à tous les points de vue du code produit. Au cours de la préparation de ma thèse [130, 131], j'ai défini les bases théoriques de ce générateur et réalisé un premier prototype de recherche (fin 1987). Puis pendant les années 88-90, je me suis concentré sur l'ensemble des modules qui réalisent les interfaces vis-à-vis des utilisateurs pour constituer la première version complète du système FNC-2 (*Frond-end* OLGA et Traducteur vers C). Puis, à partir de ce prototype (utilisable), un ensemble de travaux de recherche ont été développés lors de stages ou de thèses. J'ajouterai que l'objectif de la réalisation du système FNC-2 (la nécessité ou les objectifs d'une réalisation lourde d'un système logiciel sont souvent mal compris, ou preçus) était bien de faciliter par la suite, le développement de futurs travaux de recherche. Le système FNC-2 a joué pleinement ce rôle d'outil (support) de recherche dans les années qui ont suivi la sortie du premier prototype (1990).

À partir de 1990, les langages de spécification (OLGA , ASX , ATC) de FNC-2 ont subi un ensemble d'améliorations, ou d'importantes modifications dans le but de répondre aux demandes des utilisateurs (passerelle avec le système CENTAUR et participation dans le projet COMPARE) et pour la facilité d'expression des GAs. Cet ensemble de travaux recherchait essentiellement à unifier les domaines sémantiques (régles sémantiques) et syntaxique (grammaires) qui historiquement étaient séparés. Cela a nécessité de modifier une bonne partie des modules du système FNC-2 . Je donne ici une liste non-exhaustive de ces différentes modifications :

- élimination de la séparation entre le domaine syntaxique (les arbres) et le domaine sémantique (les attributs) (cf. la section 2.18) ;
- permettre d'utiliser une GA comme une fonction à part entière, en particulier depuis une règle sémantique d'une autre GA ;
- introduction de la notion de surcharge d'opérateur (constructeur) comme une solution au problème d'inclusion de phyla[7] (cf. les sections 2.18 et 2.19.1) ;
- introduction de la notion de nommage des champs dans une spécification ASX (cf. la section 2.19.1) ;
- introduction des notions de schéma de production et de production conditionnelle (cf. la section 2.12).

[7] non-terminaux

Comme dit précédemment, l'ensemble des travaux présentés ici sont l'œuvre d'une équipe de recherche mais malgré tout, l'implantation du système et sa conception a été presque de mon unique ressort et j'ai personnellement assuré en grande partie l'encadrement des différents stagiaires et thésards qui ont participés à son développement. Pour bien faire comprendre la cohérence de l'ensemble de mes travaux, j'ai préféré regrouper mes activités de recherche par sujets en y ajoutant les travaux des stagiaires et thésards correspondants.

2.4 Classes de grammaires attribuées

Transformation des grammaires attribuées : Classes de grammaires attribuées **Résumé des travaux [130, 131, 89]**

Pour contrer l'inefficacité en place, plusieurs classes de grammaires d'attributs qui imposent des restrictions sur la GA d'entrée ont été introduites ; grâce à ces limitations, il était alors possible de construire des évaluateurs peu gourmands en place, mais au prix d'une perte importante de puissance d'expression. Parmi ces classes, celle des GAs l-ordonnées joue un rôle particulier. En effet, elle est la plus large classe pour laquelle il est possible d'engendrer un évaluateur efficace en mémoire, — à base de séquence de visite — sans pour autant pénaliser l'efficacité en temps. Cette classe se caractérise par la connaissance statique d'un ordre total d'évaluation des attributs. En outre, toute GA peut être transformée en une GA l-ordonnée équivalente, mais avec un risque d'exponentialité de la taille de cette dernière. Enfin, le test de caractérisation de cette classe l-ordonnée est un problème NP-complet. Mon premier travail a été de concevoir un nouvel algorithme de transformation [130, 131, 91], qui fait que ce facteur d'exponentialité reste confiné à des cas pathologiques. Pour être tout à fait précis, pour l'instant, je n'ai réalisé cette transformation qu'à partir d'une sous-classe de GA nommée FNC — fortement non-circulaire — mais dans ma thèse [130], j'ai défini la transformation complète, à partir des GAs bien formées qui est la plus large classe des GAs.

2.5 Analyse de flot de grammaire

Résumé des travaux [129, 130, 89]

Cependant, cet algorithme de transformation impose d'effectuer en cascade un ensemble de tests de caractérisation (test FNC, test doublement non circulaire, etc) qui introduisaient un temps de construction important, l'autre point noir des GAs. J'ai montré que, grâce à l'introduction de nouvelles techniques [129, 130, 89] (stabilité sémantique et ordonnancement optimal), ces différents tests de caractérisation deviennent quasiment linéaires en la taille de la GA alors qu'ils sont polynomiaux en théorie. En outre, le test de caractérisation des GAs bien formées, théoriquement exponentiel, devient en pratique parfaitement réaliste [131]. Ceci m'a permis d'envisager rapidement la transformation des GAs bien formées en des GAs l-ordonnées. De plus, nous nous sommes rendus compte que ces problèmes appartenaient à une classe beaucoup plus générale — analyse de flot de grammaires — et que nos optimisations pouvaient, du même coup, prendre une signification plus importante [89].

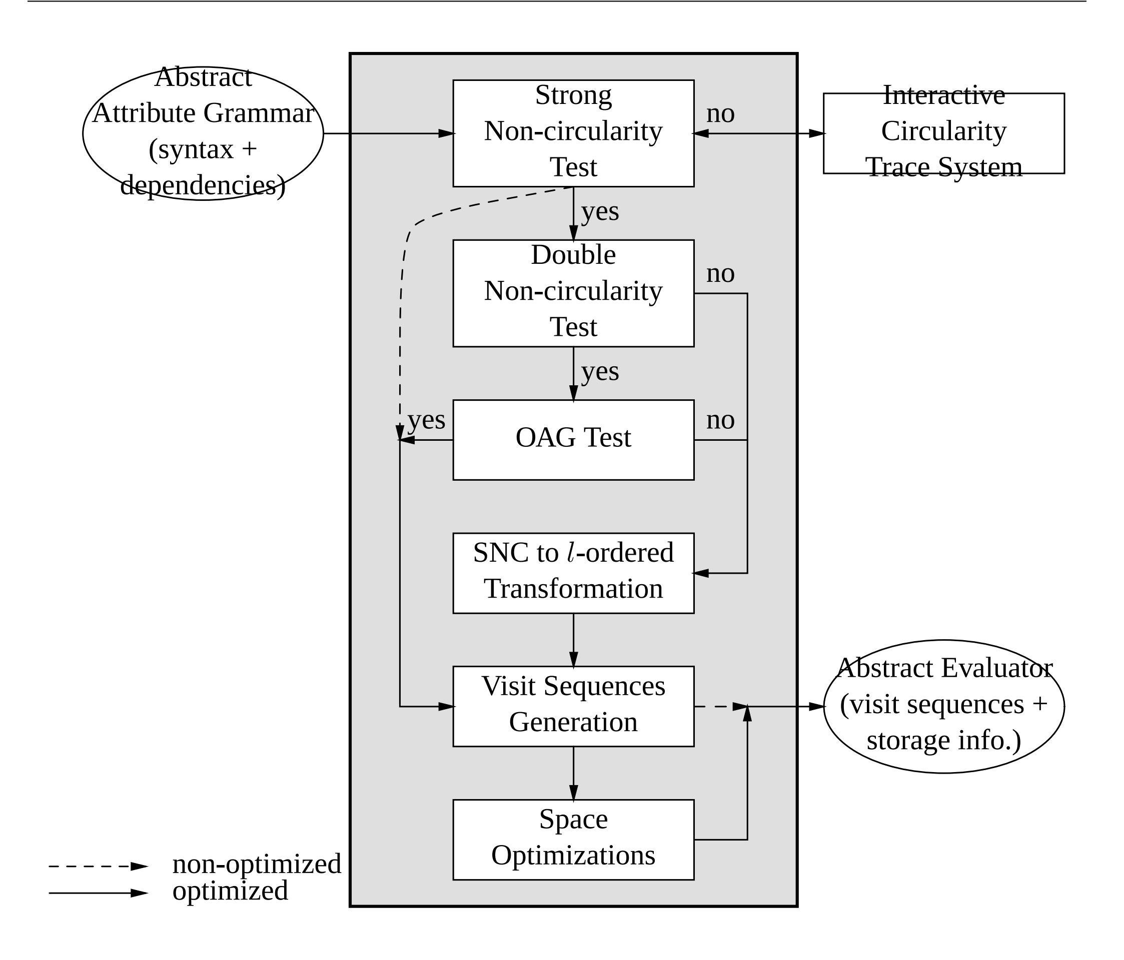


FIG. 2.3: Schéma général du générateur d'évaluateur d'attributs

2.6 Optimisation mémoire

avec C. JULIÉ (DEA [93], puis thèse [94] et les articles [95, 96])

En collaboration avec Catherine JULIÉ , nous avons travaillé sur les méthodes d'optimisation mémoire introduites sur les GAs l-ordonnées. Dans ce domaine, il existe un résultat théorique "gênant" : la recherche d'un schéma optimal de stockage des attributs devient très vite (même pour des sous-classes de GA très étroites) un problème NP-complet. Malgré cela, des méthodes ont été définies dans le cadre des GAs l-ordonnées, et donnent des résultats très satisfaisants, bien que le schéma d'allocation ne soit pas optimal. En effet, avec la connaissance de l'ordre total d'évaluation, il est possible sous certaines conditions de décider que tel attribut peut s'implanter, soit en variable globale soit en pile (c'est-à-dire hors de l'arbre). Nous avons amélioré ces méthodes au point de déterminer des conditions nécessaires et non plus seulement suffisantes. Nos résultats pratiques [94, 95] montrent qu'environ 50% à 70% des attributs sont implantés en variables, et que de 20% à 40% des attributs sont implantés en pile.

Les résultats, que nous avons acquis en 1990, nous permettent de stocker hors de l'arbre, outre tous les attributs *temporaires*, c'est-à-dire ceux dont la durée de vie est limitée à une seule visite du nœud qui les porte, une bonne partie des attributs *non temporaires* : ceux qui peuvent être stockés dans des variables globales ou dans des piles strictes [95, 96]. Nous savons aussi, au moins théoriquement, comment implanter ceux qui restent dans des piles "cactus".

Le fait de pouvoir stocker *tous* les attributs hors de l'arbre, outre ses effets bénéfiques sur la consommation de mémoire, permet d'augmenter la puissance d'expression des GAs en réutilisant les mêmes techniques d'évaluation (cf. la section 2.12). Ce travail a aussi eu de l'importance pour d'autre travaux de recherche (cf. la section 2.7 ou la section 2.13.1) et en particulier pour les GAs dynamiques.

2.7 Mises à jour destructives dans les GAs

avec ÉTIENNE DURIS **(en DEA [62])**

Une GA est une spécification déclarative d'un calcul sur un arbre qui peut être considérée comme un cas particulier de programme applicatif ou fonctionnel (sans effet de bord). Le générateur d'évaluateurs analyse la partie syntaxique de la GA et, par souci d'efficacité, la transforme en un programme impératif effectuant des effets de bord. Cependant, l'analyse des fonctions sémantiques n'est pas aussi poussée (la théorie classique des GAs n'a aucune connaisance sur les règles sémantiques) . En particulier, sous certaines conditions de typage, une règle sémantique $X.a := f(Y.a, \dots)$, où f est une fonction de mise à jour , peut être traduite par l'affectation $v := F(v, \dots)$, grâce aux analyses de durée de vie des attributs (cf. la section 2.6), qui conserve entre autres son caractère applicatif. Pour augmenter l'efficacité, on voulait remplacer cette affectation par l'instruction $P(v, \dots)$;, où la procédure P est l'équivalent destructif de F. Pour cela, il faut, d'une part déterminer les conditions dans lesquelles cette transformation est légitime et, d'autre part, pouvoir construire automatiquement P à partir de la définition de f. Nous avons étudié et résolu ces deux problèmes dans le cas où f est définie par une grammaire attribuée [62, 63]. Dans cette situation, nous avons beaucoup plus informations sur la forme des règles sémantiques. Cependant, notre algorithme déterminant la légalité de la transformation, qui repose sur la possibilité que l'ancienne valeur de v soit partagée par une valeur encore active, est assez pessimiste (bien que parfaitement sûr) et peut être amélioré.

2.8 Évaluation incrémentale

Résumé des travaux [130, 131, 91])

J'ai aussi développé une méthode originale d'évaluation incrémentale [130, 131, 91]. Quelques mots sur le problème à résoudre : soit un arbre complètement décoré par des attributs, tel que les attributs soient consistants vis-à-vis des règles sémantiques de la GA qui les définissent. Une modification syntaxique de l'arbre (comme on peut en faire avec un éditeur structurel tel que CENTAUR) peut se ramener au remplacement d'un sous-arbre consistant par un autre sous-arbre extrait précédemment d'un autre arbre lui aussi consistant. Au point de modification, cependant, les valeurs des attributs peuvent ne pas être consistantes. L'évaluation incrémentale a pour but de rétablir la consistance du nouvel arbre en ne recalculant que les instances d'attributs nécessaires. À la suite des travaux de T. REPS, ce thème de recherche a connu un grand succès. La plus grande difficulté consiste à trouver des algorithmes de réévaluation asymptotiquement optimaux, c'est-à-dire dont la complexité est proportionnelle au nombre d'instances d'attributs dont la valeur dans le nouvel arbre est différente de celle dans l'arbre initial. Le problème est que les dites instances ne sont pas connues au moment où l'algorithme commence. Dans la réévaluation incrémentale comme dans l'évaluation exhaustive, il s'agit de trouver le meilleur compromis entre l'efficacité et la puissance d'expression. Ma méthode, fondée sur des travaux de G. FILÉ [68], utilise un évaluateur quasi-déterministe capable de commencer son évaluation en tout nœud de l'arbre et de "remonter" si besoin est. Un tel évaluateur peut être construit pour toute grammaire doublement non circulaire (DNC). La classe DNC est intermédiaire entre la classe FNC et la classe l-ordonnée, mais notre transformation FNC-l-ordonnée (cf. la section 2.4) permet d'appliquer ma méthode à toute grammaire FNC. La réévaluation incrémentale s'obtient en ajoutant à cet évaluateur un contrôle sémantique, dont le rôle est de tester, avant de réévaluer une instance, si c'est effectivement nécessaire et, après avoir réévalué une instance, de comparer sa nouvelle valeur avec son ancienne, de façon à savoir s'il faut réévaluer ses descendants. J'ai pu tester notre méthode dans des conditions réalistes (cf. la section 2.8.1) dans un environnent interactif comme le système CENTAUR . De plus, la séparation entre le contrôle sémantique et la méthode de réévaluation proprement dite induit une très grande souplesse d'utilisation de cette méthode. En effet, le contrôle sémantique, en règle générale, dépend fortement du type de l'application décrite. Ainsi ce contrôle peut être étendu en fonction du type l'application traitée (voir la section 2.8.1).

2.8.1 Évaluation incrémentale dans le cadre CENTAUR

avec CHRISTOPHE ROUDET **(DEA)**

En 1994 , j'ai pu poursuivre notre travail d'intégration de FNC-2 et du système de construction d'environnements de programmation CENTAUR [31] développé par le projet CROAP à l'INRIA–Sophia. En effet, grâce au travail de Christophe ROUDET [141], encadré par Isabelle ATTALI, j'ai pu valider nos évaluateurs incrémentaux (cf. la section 2.8)

sur une application complète. Le travail de Christophe Roudet consistait à développer un générateur de décompilateurs d'arbres graphiques, dont le langage d'entrée est un sous-ensemble de PPML et qui utilise comme moteur la boîte graphique FIGUE. Grâce à l'emploi de la technologie GA (Fnc-2 /Lisp et son interface avec Centaur), les décompilateurs produits peuvent fonctionner de manière incrémentale, ce qui améliore le confort de l'utilisateur dans des applications interactives. De plus, Christophe Roudet a utilisé la grande souplesse d'utilisation induite par la séparation entre le contrôle sémantique et la méthode de réévaluation (cf. la section 2.8). Enfin il s'est très largement inspiré de notre travail sur ppat (cf. la section 2.19.2), en utilisant la même méthode et architecture logicielle.

Remarque

On peut voir cette technique de programmation comme une programmation par aspects. En effet, l'implémentation en Lisp (pour le système Centaur) utilisait fortement cette technique de code avant et après l'appel d'une méthode (règle sémantique dans notre cadre). Le contrôle sémantique spécifique à l'évaluation incrémentale correspond à enrichir le code des règles sémantiques par un code particulier avant et après l'évaluation de la règle.

2.9 Évaluateurs d'attributs sur machines parallèles

avec B. MARMOL **(DEA [114], puis thèse [115] et l'article [87])**

L'efficacité des évaluateurs séquentiels à base de séquences de visites est due à la connaissance d'un ordre *total* d'évaluation des attributs de chaque non-terminal, qui fait que l'évaluateur sait à tout instant quels attributs il a déjà calculés. En ce sens, cette contrainte est "utile" puisqu'elle permet de gagner en efficacité. En revanche, l'ordre de calcul des attributs d'une production n'est contraint que par l'ordre de calcul des attributs de chaque non-terminal et par les dépendances locales ; ceci ne définit en général qu'un ordre partiel, et tous les ordres totaux compatibles avec celui-ci sont équivalents en efficacité.

On peut donc imaginer que, sur une machine parallèle, il est important de conserver un ordre total pour évaluer les attributs de chaque non-terminal, de façon à éviter des attentes et des synchronisations inutiles, mais qu'il est bénéfique de parcourir en parallèle des sous-arbres disjoints pour calculer leurs attributs ([114]).

Nous avons créé une version modifiée de FNC-2 qui produit des évaluateurs parallèles fondés sur la méthode précédente. Ces évaluateurs sont écrits en C pour une machine Sequent Balance, dont nous possédions un exemplaire à Rocquencourt, et son système d'exploitation Dynix. L'architecture de cette machine, multi-processeur à mémoire partagée, est parfaitement adaptée à l'implantation de notre méthode d'évaluation. Après avoir résolu de nombreux problèmes très irritants dûs à notre inexpérience de la programmation d'applications parallèles (le travail de B. MARMOL), nous avons pu mener quelques expériences sur des GAs et des textes sources relativement réalistes. Les résultats sont tout à fait satisfaisants puisque nous observons une parallélisation très réussie (multiplication de la vitesse de traitement par un facteur supérieur à 5 avec 6 processeurs, soit une efficacité de 86%). Ces travaux sont décrits plus en détail dans [115, 87].

Par ailleurs, nous avons étendu les techniques d'optimisation de la gestion de la mémoire des évaluateurs classiques à nos évaluateurs parallèles. Plus précisément, nous considérons les évaluateurs parallèles comme une famille d'évaluateurs exhaustifs séquentiels, sur chacun desquels nos algorithmes d'optimisation mémoire peuvent s'appliquer. La description formelle de cette décomposition est achevée [115] s'appuyant fortement sur nos travaux de la section 2.6 ; malheureusement, nous n'avions pas le temps de la mettre en œuvre.

Dans [147], l'exemple de base et l'abstraction employée pour leurs analyses de programmes fonctionnels (graphe de dépendance) montrent assez clairement qu'il doit exister de grandes similitudes entre les méthodes de parallélisation implicite de programmes fonctionnels et celles introduites pour les grammaires attribuées. Sur l'exemple de base de cet article, notre transformation FNC- l-ordonnée effectue une transformation du programme fonctionnel qui semble être l'information recherchée par leurs analyses [147] .

Remarque

Il serait intéressant de reprendre ce travail avec les techniques d'aujourd'hui (processus léger en Java, par exemple), en particulier avec les possibilités du « GRID computing ».

2.10 La composition descriptionnelle (ou méta-composition) des GAs

Cet axe concerne la composition descriptionnelle (ou méta-composition) des GAs, définie par GANZINGER et GIEGERICH [75] avec leurs grammaires couplées par attributs. L'idée de base est que, si l'on a deux GAs spécifiant chacune une transduction d'arbres attribués (c'est-à-dire une fonction d'une première syntaxe abstraite attribuée vers une seconde), et si la syntaxe de sortie de la première est égale à la syntaxe d'entrée de la seconde, il est possible de composer ces GAs. Une manière simple pour ce faire est de produire un évaluateur pour chaque GA et de passer l'arbre de sortie du premier, qu'on construit explicitement, comme arbre d'entrée du second. H. GANZINGER et R. GIEGERICH ont prouvé qu'il existait une manière plus efficace d'aboutir au même résultat en produisant, à partir de ces deux GAs, une troisième GA spécifiant la composition des transductions sans construire explicitement l'arbre intermédiaire. De plus, ils donnent un algorithme pour effectuer cette construction.

2.10.1 Coupleur statique et dynamique

avec GILLES ROUSSEL **(Magistère [143], puis thèse [145] et les articles [146, 144])**

Nous (essentiellement le travail de Gilles ROUSSEL) avons réalisé un coupleur de GA qui, à partir de deux GAs composées séquentiellement, produit une GA qui réalise la même fonction que l'exécution séquentielle des deux GAs initiales mais sans construire l'arbre intermédiaire. Cela favorise évidemment l'écriture modulaire de grosses applications sans nuire à l'efficacité (au contraire). Notre coupleur est fondé sur l'algorithme de méta-composition de GANZINGER et GIEGERICH, modifié pour traiter les constructions “inhabituelles” d'OLGA (nœuds, listes, . . .) et les conditionnelles. Nous avons, en outre, réalisé un optimiseur de GA qui transforme une GA en une autre GA équivalente mais plus “efficace”, en particulier en court-circuitant des chaînes de règles de copie [142] ; cet optimiseur est particulièrement utile sur les GAs produites par le coupleur.

Nous proposons aussi [146, 145, 144] une autre approche de la méta-composition, plus dynamique, qui conserve l'avantage essentiel de ne pas construire les arbres intermédiaires. Cette approche a été motivée et guidée par les contraintes de l'utilisation de la méta-composition dans la mise en œuvre de la généricité dans les GAs (cf. la section 2.11). En effet, nous avons défini une méta-composition qui travaille directement au niveau des évaluateurs et non plus au niveau “source” des GAs. Les avantages de cette approche est qu'elle permet, en premier lieu, de s'affranchir des conditions de clôture de la méta-composition classique : il est en effet toujours possible de coupler deux évaluateurs selon notre méthode, alors que la méta-composition des GAs correspondantes peut être impossible pour des questions de circularité ou de classe de la GA résultante.

Plus précisément, nous avons définis deux types approches : l'une qui produit des coupleurs dit statiques (l'approche de Gilles ROUSSEL) et l'autre des coupleurs dit dynamiques (mon approche).

De plus, mon approche permet une réelle compilation séparée. En effet, nous montrons comment construire séparément des évaluateurs particuliers qui peuvent s'affranchir de l'existence physique de leur arbre de base mais accèdent au travers les uns des autres à l'arbre d'entrée de la grammaire attribuée qui fait appel à ces évaluateurs.

Nous pensions étendre cette approche en faisant en sorte que ce type d'évaluateur puisse commencer son évaluation à n'importe quel nœud de l'arbre. Ceci serait certainement fait en utilisant la trame des évaluateurs incrémentaux DNC (cf. la section 2.8) que nous avons développé dans le contexte de l'intégration de FNC-2 et de CENTAUR (cf. la section 2.8.1).

2.11 Généricité dans les Grammaires Attribuées

avec C. LE BELLEC **(DEA [105], puis thèse [106] et l'article [107])**

Une GA générique est une GA normale qui décrit un algorithme classique (comme l'identification à l'aide d'une table des symboles) sur une syntaxe très abstraite, minimale, qui ne comporte que les notions syntaxiques utiles pour cet algorithme (comme la déclaration d'un identificateur, l'utilisation d'un identificateur et le bloc). On voudrait alors avoir un moyen d'appliquer cet algorithme sur une syntaxe réelle décrivant un langage complet. Cette application est spécifiée par l'association à chaque non-terminal de la GA générique d'un ensemble de non-terminaux de la GA réelle. Il est alors possible, moyennant certaines conditions, de plaquer les calculs spécifiés par la GA générique sur la grammaire réelle et de construire une GA sur cette grammaire réelle qui est équivalente à la GA générique (notion d'instanciation).

Pour construire alors, sur la syntaxe réelle, une nouvelle grammaire attribuée réalisant la sémantique de la GA générique (notion d'instanciation), on procède ainsi :

- On construit d'abord une première grammaire qui a pour base la syntaxe de la grammaire réelle et qui fournit, pour tout arbre de celle-ci, l'arbre de la syntaxe de la GA générique qui lui correspond selon l'association entre non-terminaux définie ci-dessus.
- On compose ensuite cette grammaire avec la GA générique, à l'aide du coupleur décrit dans la section 2.10 basé sur la composition descriptionnelle.

Le problème se "réduit" donc à construire automatiquement cette première GA à partir de la syntaxe du gène (GA générique), de la syntaxe "cible" et de leur association. Nous avons mis au point une famille d'algorithmes qui effectuent cette construction dans diverses situations de complexité croissante [107, 106].

Il faut savoir que l'idée d'utiliser la méta-composition pour faire de la généricité est la pièce maîtresse de cette méthodologie que je nommerai par la suite *généricité structurelle*. J'ai personnellement eu cette idée (très simple) au tout début 1992 mais malheureusement quelque semaines plus tard dans [67] nous découvrions quasiment la même approche.

Dans la thèse de Gilles ROUSSEL [144], cette idée a été étendue par la notion de grammaires attribuées incomplètes qui essaaient d'introduire une notion de composant, module avec des attributs d'entrée et de sortie.

2.11.1 Olga ++

avec C. LE BELLEC **(DEA [105] et thèse [106])**

L'aspect intéressant d'OLGA qui met en jeu de façon importante l'analyse statique est la possibilité offerte au programmeur d'"oublier" de définir certaines règles sémantiques et de laisser le système les construire automatiquement. Cette facilité, quand les règles à construire sont limitées à des règles de copie, est connue depuis longtemps. De plus, cet ensemble de règles représente plus de 70% des règles sémantiques d'une grammaire attribuée.

En 1990 avec C. LE BELLEC, nous avons donc défini et implanté dans FNC-2 une extension d'OLGA, [105] qui consiste à définir des classes d'attributs et à donner, pour chacune de ces classes, un ensemble de *modèles* de règles sémantiques susceptibles de définir tout attribut qui lui appartient. En outre, chacun de ces modèles de règles sémantiques est subordonné à un modèle de production. Lorsqu'il s'agit de construire une règle sémantique manquante pour un certain attribut sur une certaine production, le système commence par déterminer quels sont les modèles de production qui correspondent à celle qu'il considère, puis essaie d'instancier les modèles de règles sémantiques qui lui sont attaché dans le contexte de cette production. Si cela est possible pour un et un seul de ces modèles, la règle sémantique correspondante est effectivement créée.

Remarque
Ce travail peut être comparé à la notion de *template* de C++ avec une notion héritage entre *template*, mais aussi aux langages de point d'ancrage introduis dans la programmation par aspect [4]. En effet dans Olga++, le langage de pattern (partie gauche) d'Olga++ peut être vu comme un langage pour définir les points d'ancrage possibles des règles sémantiques.

2.11.2 Forme normale d'une grammaire attribuée

avec LOÏC CORRENSON en stage d'option de l'École Polytechnique [45]

Nous avons implanté dans le système FNC-2 nos travaux de 1993-1994 sur la généricité dans les grammaires attribuées, en réalisant un “instancieur” de GA. Celui-ci prend en entrée un gène, c'est-à-dire une GA décrivant un algorithme sur une syntaxe minimale, une syntaxe cible et une description de l'association entre les éléments des deux syntaxes, et produit une nouvelle GA qui décrit l'exécution de l'algorithme du gène sur la syntaxe cible.

Au cours de cette réalisation, L. CORRENSON a amélioré l'applicabilité de la méthode en introduisant la notion de *forme standard* de la syntaxe d'une GA, ainsi qu'un algorithme de réduction d'une GA en sa forme standard qui préserve la sémantique. Il a aussi prouvé qu'un gène transformé en sa forme standard est instanciable sur un nombre au moins aussi élevé (et en général plus élevé) de syntaxes cibles que lorsqu'il est sous sa forme originelle. Il est à noter que cette notion de forme standard peut aussi servir à déterminer (avec des critères syntaxiques) si deux GA sont équivalentes. Par ailleurs, il a donné une nouvelle définition, plus large, de la relation de correspondance entre deux syntaxes qui prend en compte mon travail de la section 2.18 sur la notion de sous-typage introduit par le langage TYPOL (Sémantique Naturelle) dans CENTAUR. Ces travaux sont décrits dans [45] et ont valu à Loïc CORRENSON le prix d'option de l'École Polytechnique.

2.12 Les Grammaire Attribuée Dynamiques : une nouvelle vision des GAs

Résumé des travaux [133, 135, 136]

Bien que les GAs aient été introduites il y a trente ans, leur manque de pouvoir d'expression les a jusqu'ici confinées dans le domaine du traitement des langages de programmation. Nous voulons étudier les conséquences de la possibilité de stocker tous les attributs hors de l'arbre (cf. la section 2.6). En effet, s'il ne sert plus à porter les attributs, le seul rôle de l'arbre est de guider le parcours de l'évaluateur. L'arbre peut alors être un objet dynamique et/ou virtuel.

Depuis longtemps, je soutiens qu'il est possible d'étendre cette expressivité et que les GAs peuvent être utilisées pour décrire des calculs sur des structures qui ne sont pas uniquement des arbres, mais aussi des formes abstraites permettant de décrire des structures infinies.

Cette nouvelle formulation est fondée sur les deux remarques suivantes :

– la théorie des GAs ne se préoccupe ni des moyens mis en œuvre pour la construction des arbres ni de leur représentation physique ;
– la théorie des GAs n'utilise pas la propriété de terminaison des calculs induite par le fait qu'ils sont dirigés par une structure finie.

Plus précisément, notre interprétation de la grammaire sous-jacente à une GA est la même que celle de la grammaire décrivant les arbres d'appels pour un programme fonctionnel donné ou les arbres de preuve pour un programme logique donné : elle décrit l'ensemble des flots de contrôle possibles. Dans ce contexte, une production décrit un schéma récursif élémentaire (flot de contrôle), tandis que les règles sémantiques décrivent les calculs associés à ce schéma (flot de données).

Il est très important de remarquer que la plus grande partie des résultats théoriques et pratiques concernant les GAs, en particulier les algorithmes de construction d'évaluateurs efficaces, sont fondés uniquement sur une abstraction du flot de contrôle grâce à la grammaire, et pas du tout sur la manière dont une instance particulière de ce flot est obtenue au cours d'une exécution. En conséquence, nous proposons [133] deux extensions syntaxiques qui sont en accord avec cette vision :

– les productions conditionnelles qui permettent de diriger le flot de contrôle grâce à des valeurs d'attributs,
– et les schémas de production qui permettent de décrire un schéma de récursion indépendamment de toute structure physique et/ou permettent une combinaison différente des éléments d'une structure physique (production) donnée.

Nous obtenons ainsi un langage dont le pouvoir d'expression est comparable à celui de la plupart des langages fonctionnels du premier ordre, avec un côté déclaratif beaucoup plus marqué.

Comme ces extensions ne remettent pas en cause les bases du formalisme des GAs, leurs implantations dans le système FNC-2 ne concernent que les parties avant et arrière du compilateur OLGA et pas le générateur d'évaluateurs, qui est la partie la plus importante et

la plus compliquée. J'ai réalisé en quelques hommes-semaines une première implantation des grammaires attribuées dynamiques.

L'intérêt de ces extensions est de redonner aux GAs leur expressivité intrinsèque. De plus, elles nous permettent d'envisager de nouveaux axes de recherche en comparant nos techniques d'analyses à celles qui ont été développées dans des formalismes de même expressivité (langage fonctionnel). C'est dans ce cadre que s'inscrivent en particulier nos travaux sur les mises à jour destructives dans les GAs [63] ou les relations avec la programmation fonctionnelle (cf. la section 2.13.1).

Notre première présentation des GAs dynamiques [133] était, volontairement, assez informelle et intuitive. En 1996, nous avons formalisé la notion de GA dynamique, ainsi que leur implantation à base de séquences de visites [135, 136] ; nous avons en particulier prouvé formellement la validité de la technique de "changement de plan", qui permet de réutiliser sans modification un générateur d'évaluateurs existant, comme celui de notre système FNC-2 .

2.13 Relation avec la programmation fonctionnelle

Résumé des travaux [64, 65, 66]

Il est intéressant de noter que, au cours de ce travail, nous avons été amenés à étudier un problème équivalent dans le cadre de la programmation fonctionnelle, ainsi que les solutions qui lui ont été apportées précédemment. Cela a renforcé notre conviction – déjà étayée par des travaux de Thomas JOHNSSON [84] et d'autres – qu'il existait des relations très étroites entre ces deux styles de programmation (GA et programmation fonctionnelle) et que l'étude de ces relations pouvait être très fructueuse pour les deux. Par exemple, nous avons constaté [62] des similarités, mais aussi des différences, entre la méta-composition des GAs et la déforestation des programmes fonctionnels, deux transformations de programme visant à éliminer la constructions de structures intermédiaires inutiles ; nous espérions, en poursuivant cette étude comparée, pouvoir améliorer chacune de ces deux méthodes.

2.13.1 Programmation dirigée par la structure et déforestation

Avec ÉTIENNE DURIS **et** GILLES ROUSSEL

Les GAs dynamiques nous permettent d'atteindre le même pouvoir d'expression que les langages fonctionnels du premier ordre. Dans ceux-ci, le problème de l'élimination des structures intermédiaires (déforestation) est abordé à l'aide de l'opérateur de contrôle générique *fold*, qui permet de rendre la structure des données plus explicite (programmation *dirigée par la structure*). Cette propriété structurelle étant intrinsèque aux GAs, nous avons été amenés à comparer les GAs et les *folds*, tant sur le plan du pouvoir d'expression que sur celui de leurs méthodes de "composition optimisante" (composition descriptionnelle et fusion, respectivement).

Dans [64, 65, 66], nous présentons nos résultats. Les *folds* du premier ordre sont équivalents aux GAs purement synthétisées. Dans ce cadre, la fusion des *folds* et la composition descriptionnelle aboutissent au même résultat, mais cette dernière est complètement indépendante de l'évaluation (transformation de source à source) tandis que la fusion nécessite la connaissance de la méthode d'évaluation. De plus, beaucoup de programmes qui nécessitent l'usage de *folds* d'ordre supérieur peuvent être traités avec des GAs du premier ordre, grâce à la notion d'attribut hérité. Les approches différentes de ce problème dans ces deux formalismes permettent d'envisager des fertilisations croisées entre GA et programmation fonctionnelle (voir la thèse [50] de LOIC CORRENSON et la sous-section 2.15).

Plus généralement, les *folds* étant des *catamorphismes* [69], nous avons étudié les GAs en termes algébriques, afin de comparer plus facilement les résultats obtenus en GA et en programmation fonctionnelle et de faciliter le passage de connaissances d'un domaine à l'autre. En particulier, nous avons étudié notre notion de généricité structurelle dans le contexte des programmes fonctionnels [66] et le résultat est que les techniques des grammaires attribuées sont plus performantes pour l'élimination des structures intermédiaires passées en argument des fonctions (définies par des attributs hérités).

2.14 Évaluation indulgente

Avec ÉTIENNE DURIS

Certains exemples de programmes fonctionnels se traitent en “une passe” en évaluation paresseuse mais nécessitent une décomposition en plusieurs fonctions en cas d’évaluation stricte. En général, cette décomposition doit être effectuée à la main. Or ces exemples s’écrivent en GA aussi facilement que la version fonctionnelle paresseuse tandis que la décomposition (calcul de l’ordre d’évaluation et des séquences de visite) est réalisée automatiquement par le système. Nous avions démarré une étude systématique sur ce sujet, mais nous n’avions pas des résultats méritant publication, juste quelques comparaisons avec le langage fonctionnel paresseux HASKELL sur des exemples types [86].

D’après l’étude de GUY TRAMBLAY [159, 158], l’évaluation indulgente (une forme particulière de l’évaluation paresseuse) semble être une forme très courante en pratique. En effet, il montre que quasiment, tous les programmes décrits comme étant paresseux dans la littérature, ne sont en réalité que des programmes indulgents. Les programmes qui restent vraiment paresseux, sont ceux qui utilisent une structure de données infinie, comme le calcul des nombres premier en utilisant une liste infinie de nombres. Les travaux sur l’évaluation indulgente (sur le langage ID[8] [148] ou encore HP[9] de [147]) sont souvent liés à l’évaluation parallèle.

[8]langage fonctionnel paresseux
[9]Haskell parallèle et paresseux

2.15 Sémantique equationnelle

Avec LOIC CORRENSON **(thèse [50] et les articles [48, 47, 49]))**

Dans la thèse de LOIC CORRENSON [50], nous avons approfondi les travaux d'Etienne DURIS sur ce parallèle entre les Grammaires Attribuées et les programmes fonctionnels. Pour cela, il a défini, un nouveau formalisme, dénommé Sémantique équationnelle, pour les grammaires attribuées, qui prend en compte nos travaux sur les grammaires attribuées dynamiques et sur l'unification du domaine syntaxique et sémantique. Avec ce formalisme il montre que tout programme fonctionnel (même d'ordre supérieur) peut être traduit en sémantique équationnelle. Dans sa thèse, il a défini une sémantique opérationnelle de cette sémantique, par un système de réécriture. Dans ce formalisme il a pu définir et étendre les transformations de programme des grammaires attribuées (composition structurelle) et montré toute leur puissance. En particulier, il a montré que nos techniques de transformation structurelle pouvaient s'appliquer aussi sur des programmes fonctionnels d'ordre supérieur.

2.16 Sémantique dénotationnelle

Avec STÉPHANE LEIBOVITSCH **(DEA [109])**

Les relations entre la sémantique dénotationnelle et les GAs ont été peu étudiées : un seul article de H. GANZIGER [74] traite vraiment de ce sujet, en montrant comment traduire une spécification de langage en sémantique dénotationnelle en un interprète écrit en GA. Malheureusement, les GAs produites dépassent le cadre traditionnel des GAs, puisqu'elles nécessitent une évaluation paresseuse et circulaire. Or, les GAs dynamiques permettent de décrire ce type d'extension. Nous avons donc étudié si elles pouvaient constituer une réponse aux diverses questions laissées ouvertes par H. GANZINGER.

S. LEIBOVITSCH a su répondre positivement à cette question [109]. Il a repris et adapté le schéma de traduction de GANZINGER et montré que les GAs qu'il produit sont des GAs dynamiques acceptables par FNC-2 . Il a, en outre, réalisé (au moins en partie) un prototype de traducteur automatique mettant en œuvre ce schéma. La seule construction qui pose encore problème est l'utilisation de continuations pour traiter les transferts de flot de contrôle non structurés (l'instruction `goto`).

2.17 Transformation d'arbres attribués

Avec A. SOUAH **(en DEA [151], puis en thèse [152])**

A. SOUAH a travaillé sur la transformation d'arbres attribuées : cette technique permet de décrire agréablement des transformations de programmes — optimisations dans les compilateurs , traduction vers un langage intermédiaire — que les grammaires attribuées ne permettent pas.

Son sujet consistait à concevoir et implanter un nouveau système de transformation d'arbres attribués (STAA) qui se distingue des travaux antérieurs [111] par sa puissance d'expression supérieure, un côté déclaratif plus marqué et une sémantique beaucoup mieux définie. On peut voir un tel système de transformations comme un système de réécriture conditionnelle où les conditions sont exprimées à l'aide d'une grammaire attribuée. Nous avons défini une notion de *cohérence* d'un tel système de réécriture conditionnelle.

Cette approche pose plusieurs problèmes théoriques :

- Au cours du processus de transformation, le contexte associé à l'arbre transformé évolue. Ainsi, il est possible que plusieurs instances d'attributs changent de valeur. En particulier, les instances d'attributs (arguments des prédicats de pré-conditions) peuvent être affectées par la modification. Donc la valeur d'une pré-condition d'une règle de transformation dont le modèle d'entrée filtre un sous-arbre peut varier. Nous dirons qu'une pré-condition est stable si sa valeur ne varie pas quelle que soit l'évolution du contexte par ailleurs. L'idée clé de notre approche est qu'une transformation ne peut s'effectuer que si sa pré-condition est dans un état stable. Nous dirons qu'une telle transformation est cohérente. Bien sûr, nos travaux sur l'évaluation incrémentale (cf. la section 2.8) trouvent, ici, une application directe.
- La notion de cohérence nous a poussés à étudier la possibilité de vérifier la cohérence d'un système de transformations de manière statique. En effet, le test de la cohérence du système revient à pouvoir calculer les dépendances entre deux instances d'attributs dans n'importe quel arbre de dérivation. Nous proposons deux algorithmes très proches des algorithmes de tests de caractérisation (cf. la section 2.4) : le premier calcule cette dépendance dans le cas des GAs bien formées, tandis que le second se restreint à la classe des grammaires Doublement Non Circulaires (DNC).
- L'aspect déclaratif du système pose deux problèmes :
  1. Du fait qu'on n'associe pas de stratégie de parcours d'arbre au système de transformation, le système peut effectuer les transformations dans n'importe quel ordre ; il faut garantir à l'utilisateur l'unicité du résultat de ses transformations : nous introduisons la notion de A-confluence.
  2. L'existence de plusieurs stratégies de transformation nous amène à déterminer la manière optimale d'opérer les transformations : il est légitime alors d'essayer de trouver un moyen pour éviter les transformations inutiles. En s'inspirant des travaux existants sur les systèmes de réécriture et en utilisant la méthode des GAs (évaluation incrémentale cf. la section 2.8), on obtient un algorithme simple trouvant la stratégie optimale.

L'essentiel de notre effort a porté sur la formalisation mathématique de cette idée de *cohérence* en terme de terminaison et de confluence. Mais il faut bien admettre le point faible de notre approche : la notion de cohérence d'un STAA est indépendante de la stratégie de transformation, ce qui conduit à rejeter beaucoup de STAA intéressants et pratiques, alors que l'utilisateur recherche une stratégie rendant le système cohérent.

Remarque

Ces travaux montrent les limites d'une approche théorique pour les outils de transformation de programme. De plus, nous pouvons faire un parallèle avec l'outil de transformation XSLT pour les documents en XML qui pourrait utiliser ces résultats pour une évaluation incrémentale.

2.18 Sémantique Naturelle

L'objectif de l'opération INTERSEM financée par le GRECO de Programmation qui s'est déroulé d'avril 1989 à décembre 1991, était de proposer une alternative intéressante au bien connu “Synthesizer Generator” [140] développé par T. REPS et T. TEITELBAUM, en réunissant les compétences des équipes de recherches françaises suivantes :

– le projet CROAP de G. KAHN avec l'environnement de programmation multi-langgages CENTAUR [31];
– l'équipe de P. FRANCHI-ZANNETTACCI avec le système GIGAS [39] à l'ESSI de Sophia-Antipolis.
– nous même, avec les grammaires attribuées et FNC-2 ;

De notre côté, nous avons essentiellement travaillé sur le couplage de FNC-2 et CENTAUR, et plus précisément sur l'intégration dans CENTAUR d'évaluateurs d'attributs exhaustifs ou incrémentaux produits par FNC-2 . Mais c'est aussi la base de notre collaboration sur la traduction de TYPOL (le langage pour la Sémantique Naturelle) en GA avec I. ATTALI [22], dans le but de proposer une évaluation incrémentale de TYPOL et aussi une meilleure efficacité en espace mémoire. Une présentation du système MINOTAUR, résultat de tout ce travail, se trouve dans [23].

Sans reprendre nos longues discutions avec I. ATTALI, la traduction de TYPOL en GA, n'a pas été aussi simple que l'on pouvait l'imaginer au départ et deux à trois années plus tard DEPHINE TERRASSE [157] a rencontré des problèmes analogues dans la conception de l'interface TYPOL et COQ (du projet COQ INRIA/Rocquencourt).

L'un des plus importants a été certainement la notion d'*inclusion de phylum* (une notion de sous-type), surtout que le typage était, aussi bien pour le langage METAL que pour le langage TYPOL (traduit en prolog), un mécanisme dynamique (résolu à l'execution). Mais je dois tout de même admettre que cela a permis de faire évoluer avantageusement mes travaux de recherche sur les GAs (unification du domaine sémantique et syntaxique) et le développement de FNC-2 .

Ce travail se décompose en les points suivants :

– traduction d'OLGA en Le_LISP ;
– génération d'évaluateurs incrémentaux ;
– évolution de nos propres langages de spécification pour unifier les différentes notions (syntaxe abstraite) entre les deux systèmes ;
– travail de formulation en terme de grammaires attribuées de la notion de sous-typage utilisée aussi bien dans les spécifications de syntaxe abstraite METAL qu'en sémantique naturelle (TYPOL).
– réalisation d'un environnement CENTAUR pour l'utilisation de FNC-2 comme outil de démonstration.

2.19 Quelques autres travaux

Dans cette section, je citerai uniquement que deux travaux qui me semble importants pour expliquer ma démarche dans le développement de FNC-2. Mais pour une vision complète de mes travaux, le lecteur pourra se référer à la liste des stages (DEA, DESS) (cf chapitre 5) et des projets de recherche que j'ai encadrés.

2.19.1 Le projet ESPRIT COMPARE

Le projet ESPRIT 5399 COMPARE (COMpiler generation for PARallel machinEs) a démarré le 01.01.1991. Outre nous-mêmes, ce projet regroupait le CWI (Pays-Bas), le GMD (R.F.A.), Associated Computer Exports ACE (Pays-Bas, coordinateur), Harlequin (Royaume-Uni), STERIA (France) et, en tant que partenaire associé, l'Université de Sarre (R.F.A). Il disposait d'une force de travail d'environ 100 hommes / années sur quatre ans, et la part de l'INRIA correspondait à **trois personnes à temps plein**.

Le but de COMPARE était de développer un certain nombre de générateurs produisant, à partir de spécifications de haut niveau, des (parties de) compilateurs pour des langages plus ou moins classiques et des machines disposant d'un certain degré de parallélisme (RISC et superscalaires essentiellement). Ces moteurs communiquent entre eux à travers une représentation intermédiaire et sont dirigés par un superviseur ; la RI (représentation intermédiaire), son interface vis-à-vis des moteurs et le superviseur forment ensemble le système de compilation (COSY) de COMPARE . COSY autorise une très grande diversité dans la conception et la réalisation de la communication entre les divers moteurs, depuis un enchaînement séquentiel sur un mono-processeur jusqu'à une coopération entre moteurs incrémentaux sur une machine parallèle. Le lecteur trouvera une bonne et courte présentation de COSY dans [44].

Notre participation à COMPARE a consisté essentiellement en l'adaptation de nos systèmes PAGODE (un méta-générateur de code) et FNC-2 à COSY et, en particulier, à son langage de description de RI (représentation intermédaire), fSDL [42] (langage de définition des structures), et à la CCMIR (Common Compare Medium-level IR), ainsi qu'en la production, à l'aide de PAGODE [61, 43] d'un générateur de code pour le processeur SuperSPARC. Je noterai que FNC-2a été choisi par le *Consortium* COMPARE [41] comme le meilleur système de grammaires attribuées parmi d'autres systèmes concurrents.

Remarque

Ces travaux ont été, dans un certain sens, précurseurs dans ce domaine pour essentiellement les raisons suivantes :

1. Il existe de grand similitude entre ce langage fSDL et les Schema pour XML. J'ai retrouvé dans les Schema XML, diverses notions d'extension, d'héritage pour définir des structures de données.
2. La plate-forme COSY était une première expérience d'architecture par composants pour un compilateur, avec un langage spécifique de coordination.

2.19.2 Le générateur de décompilateurs PPAT

avec J.P JOUVE (stage d'option de l'École Polytechnique [92])

PPAT est un générateur de décompilateurs d'arbres attribués similaire à PPML (celui de CENTAUR) mais capable en plus de traiter les valeurs d'attribut associées aux noeuds. La conception de PPAT avait été l'objet du stage de DEA d'E. PLANES [138] en 1989. Pendant l'année 90, J.P JOUVE en a fait une réalisation complète [92]. Outre l'utilité de PPAT en tant que tel, cette réalisation a été très enrichissante pour nous car elle constitue l'une des premières grosses applications de FNC-2 à avoir été effectuée par un utilisateur extérieur au petit cercle des développeurs de FNC-2 . Ce fut un succès total : en trois mois et alors qu'il ne connaissait pas FNC-2 du tout, J.-P. JOUVE a écrit, mis au point et testé plus de 8000 lignes d'OLGA et autres langages, réparties en plus de 15 fichiers. Il a ainsi validé l'utilisabilité du système et les concepts dont nous nous étions fait le héraut : modularité, réutilisabilité, etc. La figure 2.4 montre le schéma général pour cette application (compilateur PPAT).

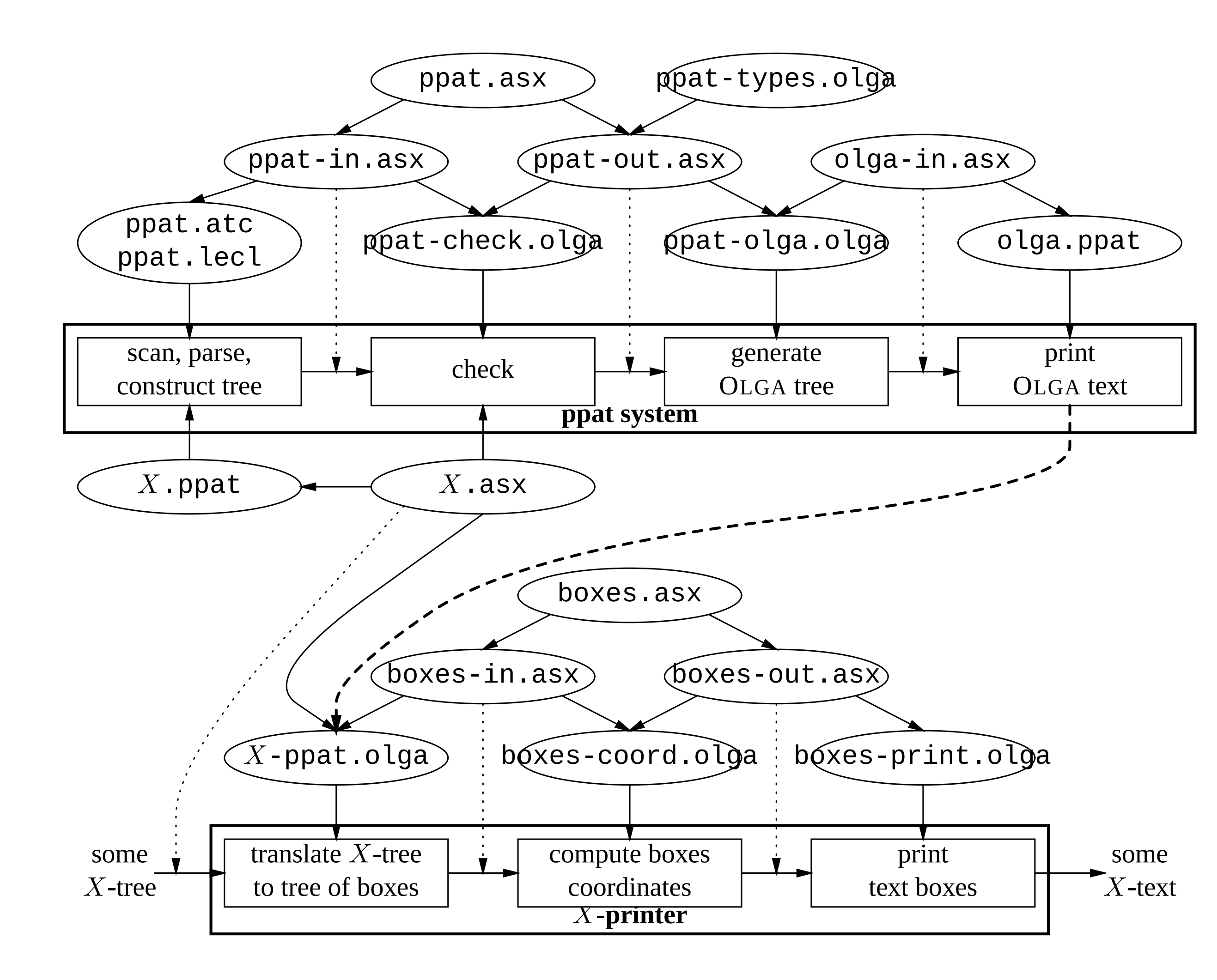


FIG. 2.4: Schéma général de PPAT

2.20 Conclusion

Pour conclure sur les qualités du générateur d'évaluateurs, j'énoncerai deux remarques sur notre propre expérience en tant qu'utilisateur du système (l'ensemble de la carrosserie, le *front-end* du langage OLGA et les traducteurs — C, Le_LISP, fSDL, Caml — sont spécifiés en GA :

– Lors d'une génération du système, c'est la compilation des objets C engendrés qui prend le plus de temps, et non leur construction proprement dite. Ceci montre une très grande efficacité en temps, d'une part lors de la construction des évaluateurs, et d'autre part lors de leur exécution.
– Enfin, depuis la sortie de la première version de FNC-2 en 1990, il a été fortement utilisé au sein de notre projet (environ 75 000 lignes de code pour le développement

autour de FNC-2 et 50 000 pour le projet ESPRIT COMPARE).

En conséquence, j'ai encadré personnellement de nombreux utilisateurs et même s'ils ont accaparé beaucoup de mon temps, ils m'ont permis de tester FNC-2 en profondeur et de confirmer nos choix sur le langage OLGA. En effet, il apparaît que la majorité de leurs erreurs sont détectées à la construction (statiquement), et celles qui peuvent rester sont dûes principalement à leur manque d'expérience d'utilisation dues langage OLGA (surtout sur les fonctionnalités peu classiques du langage).

Nous sommes convaincus que nous avions atteint, à travers la réalisation de FNC-2 , notre objectif de départ. En effet, les très bonnes qualités de la méthode des GAs — spécifications déclaratives et structurées — étaient masquées par les problèmes d'exécution et de langage de spécification qui n'avaient pas reçu jusqu'ici de solution satisfaisante. Notre système propose donc une solution satisfaisante, comme le montrent nos propres résultats, et permet donc enfin, de rendre à la programmation par GA ses qualités.

Pour conclure, je ferai référence à l'excellent *survey* sur les grammaires attribuées [124] où l'auteur mentionne très explicitement les grandes qualités de notre système FNC-2 et les motivations de réaliser un tel système pour promouvoir les GAs.

Troisième partie

Le générateur d’environnement : SMARTTOOLS

Avant propos

Depuis 2000, l'outil SMARTTOOLS est développé de manière incrémentale et a fortement évolué passant d'une version proche de CENTAUR[31, 155] (version 1) avec aïoli [30], VTP (*Virtual Tree Processor*), PPML [30] (*Pretty Printing Meta Language*) à une version ouverte basée sur des techniques non-propriétaires et sur les technologies XML et donc n'ayant plus rien de commun avec CENTAUR.

Nous pouvons lister les quatre étapes importantes suivantes dans la conception de SMARTTOOLS :

- Etape 1 : première version très proche de l'outil CENTAUR ;
- Etape 2 : construction d'une architecture centrée autour d'un bus logiciel en utilisant une communication à base de messages asynchrones ;
- Etape 3 : passage aux technologies XML (structure d'arbre et utilisation de XSLT pour les outils de visualisation, le langage Xpp) ;
- Etape 4 : une architecture à base de composants et le langage COSYNT.

Chapitre 3

Présentation générale de SMARTTOOLS

3.1 Introduction

La qualité du logiciel et sa capacité à évoluer, ainsi que la rapidité du développement, sont des soucis majeurs pour les industriels. Un logiciel bien conçu doit pouvoir s'adapter rapidement aux demandes des clients et aux nouvelles technologies pour pouvoir lutter contre la concurrence. Il doit aussi être capable d'échanger des données très variées avec d'autres applications, particulièrement depuis l'avènement d'Internet.

Adopter des formats de données standardisés facilite l'échange d'informations entre logiciels. Le W3C (*World Wide Web Consortium*) [11] élabore des spécifications[1] pour les formats de données (XML - *eXtensible Markup Language*), les langages (XSL - *eXtensible Stylesheet Language*, SVG - *Scalable Vector Graphics*) et les protocoles (SOAP - *Simple Object Access Protocol*) liés à Internet. Il donne aussi la possibilité aux concepteurs de décrire les structures des données échangées en utilisant les formalismes DTD (*Document Type Definition*) ou Schema. Les concepteurs définissent des langages dits métiers (par opposition aux langages de programmation) très variés et liés à un domaine d'application : télécommunications, mais aussi finance, assurances, transports, etc. Toutes les techniques

[1] Les spécifi cations du W3C (XML et DTD, DOM, XSL et XSLT, Schema, BML, MathML, SVG, XPath, XHTML, SOAP et WSDL) sont accessibles sur le site du W3C (`http ://www.w3c.org`).

liées aux langages de programmation peuvent être employées pour les langages métiers d'autant plus que ces derniers ont souvent une syntaxe et une sémantique plus simples. Mais les concepteurs et les utilisateurs de langages métiers n'ont pas forcément de compétences approfondies sur les techniques issues de la programmation (analyse, compilation, interprétation, etc). Il y a donc un besoin d'outils pour faciliter l'utilisation de ces techniques. De plus, de telles applications (liées à l'Internet) nécessitent un développement rapide, des possibilités d'intégration, une utilisation facile et un affichage multi-supports.

Les objectifs de la plate-forme SmartTools s'inscrivent parfaitement dans cette nouvelle problématique de conception rapide et simplifiée de langages métiers pour l'échange et/ou le traitement d'informations. Plus précisément, à partir d'une description d'un langage (DTD ou Schema), la plate-forme génère un environnement de développement contenant un analyseur d'une forme concrète du langage (*parser*), l'afficheur associé (*pretty-printer*), un éditeur syntaxique et un ensemble de fichiers Java facilitant l'écriture de traitements sémantiques (analyses, transformations).

L'originalité et l'innovation de notre approche peuvent se synthétiser en cinq points qui seront développés dans les sections suivantes :

1. Accepter en entrée des formats non propriétaires définis par le W3C (DTD et Schema) et profiter ainsi des nombreux développements réalisés autour de XML. Ainsi, le coût et le temps de développement de l'outil peuvent être fortement réduits. Notre innovation consiste à proposer des traitements sur des documents XML, en utilisant une méthodologie de programmation basée sur le patron visiteur (*visitor design pattern*) [73, 127, 126], issu de la programmation par objets.
2. Proposer une programmation par aspects [98, 101, 33] au-dessus de la technique des visiteurs ne requérant pas de transformation de code. Cette approche dynamique a l'intérêt d'être beaucoup plus simple dans sa mise en œuvre que les approches plus classiques et généralistes [110]. Mais surtout, elle aura certainement un grand intérêt dans le cadre d'applications Web pour traiter les problèmes de reconfiguration, d'adaptation et de sécurité des composants.
3. Posséder une architecture logicielle modulaire [28] (avec des composants indépendants) et extensible pour assurer une bonne évolution de l'outil. Nos choix ont été confirmés avec la réalisation quasiment naturelle d'une version répartie et surtout par la facilité d'ajout de nouveaux composants et d'interconnexion avec d'autres plates-formes comme par exemple .NET [119], avec le protocole SOAP.
4. Fournir une interface utilisateur conviviale. L'innovation de notre approche consiste à traiter tous les aspects d'affichage, y compris l'interface utilisateur, selon le même modèle. Il se dégage ainsi une approche homogène et uniforme ayant un fort potentiel de réutilisation tant pour SmartTools que pour les environnements produits. Un autre avantage important est que les techniques utilisées permettent d'exporter les vues graphiques vers d'autres supports dont le Web.
5. Auto-utiliser l'outil pour le développer ; ainsi les techniques proposées sont directement testées. Par exemple, tous les langages de description propres à SmartTools ont été développés grâce à l'outil. Chaque environnement produit réutilise les composants de SmartTools.

Cet article souhaite montrer les passerelles établies dans SmartTools entre différentes familles technologiques (langages, objets, XML), et comment l'adéquation de ces technologies a permis de construire un système ouvert et évolutif. L'innovation de notre système vient principalement de leur mise en commun. C'est dans ce sens qu'il n'existe pas de système comparable à SmartTools même si chaque élément pris séparément se retrouve dans bien d'autres outils ou travaux de recherche (générateurs d'environnements [31, 102, 140], compilateurs [91, 166] pour Java [72, 125]). Cet article justifie notre démarche en présentant les avantages et avancées obtenues par la mise en commun de ces technologies. Il insiste sur l'intérêt d'utiliser les technologies XML et les composants existants dans le cadre d'un tel développement. Ce travail peut être considéré comme les prémices d'une utilisation des technologies objets et langages pour le Web Sémantique, en particulier les travaux sur la sémantique des langages de programmation.

L'article se décompose en six sections. La première section introduit les formalismes de base (syntaxe abstraite), les liens avec les formalismes équivalents du W3C et les outils associés (éditeur structuré). La deuxième section présente les outils pour la programmation des traitements sémantiques comme la programmation par visiteur ou la programmation par aspects. La troisième section donne un aperçu de l'architecture du système organisée autour d'un bus logiciel (contrôleur de messages). La quatrième section décrit notre approche uniforme pour la conception et la réalisation des interfaces graphique et de l'interface utilisateur de SmartTools. La cinquième section présente quelques applications de notre outil. La sixième section compare notre approche vis-à-vis de travaux similaires et décrit les perspectives de nos travaux.

3.2 Syntaxe abstraite et outils

Tous les outils de SmartTools sont basés sur la notion de syntaxe abstraite étendue et fortement typée (AST[2] - *Abstract Syntax Tree*) que nous allons définir dans cette section. Cette notion de syntaxe abstraite est bien connue et est couramment utilisée dans de nombreux générateurs d'environnements ou de compilateurs [31, 91, 102]. Cette section décrit le langage de définition d'AST, l'implantation des arbres manipulés, les passerelles réalisées pour importer d'autres formats de définition d'AST et enfin les différents outils générés.

3.2.1 Langage de définition de syntaxe abstraite

Les concepts importants de la définition d'une syntaxe abstraite sont les constructeurs (opérateurs) et les types. Les constructeurs sont regroupés dans des ensembles nommés : les types. Les fils (paramètres) des constructeurs sont typés. La partie gauche de la Figure 3.1 montre la définition incomplète de notre langage jouet : `tiny`[3]. Par exemple, le constructeur `affect` est de type `Statement` et possède deux fils : le premier de type `Var` et le second de type `Exp`.

Il existe trois catégories de constructeurs :

[2] Dans le reste de cet article, nous utiliserons cette abréviation pour désigner un arbre de syntaxe abstraite.
[3] Langage utilisé comme fi l d'Ariane au cours de cet article.

```
Formalism of tiny is
Root is %Top;

Top = program(Decls declarations,
              Statements statements);
Decls = decls(Decl[] declarationList);
Statements = statements(Statement[]
                        statementList);
Statement =
  affect(Var variable, Exp value),
  while(ConditionExp cond,
        Statements statements),
  if(ConditionExp cond,
     Statements statementsThen,
     Statements statementsElse);
Var = var as STRING;
Exp =
   %ArithmeticOp,
   var,
   int as STRING,
   true(),
   false();
...
End
```

```
<!ENTITY % Top 'program'>
<!ENTITY % Decls 'decls'>
<!ENTITY % Statements 'statements'>
<!ENTITY % Statement 'if|while|affect'>
<!ENTITY % Var 'var'>
<!ENTITY % Exp 'false|int|var|true|
                 %ArithmeticOp;'>
<!ELEMENT program  (%Decls;,
                    %Statements;)>
<!ELEMENT decls  (%Decl;)*>
<!ELEMENT statements  (%Statement;)*>
<!ELEMENT affect  (%Var;, %Exp;)>
<!ELEMENT while  (%ConditionExp;,
                  %Statements;)>
<!ELEMENT if  (%ConditionExp;,
               %Statements;,
               %Statements;)>
<!ELEMENT var    (#PCDATA)>
<!ELEMENT int    (#PCDATA)>
<!ELEMENT true EMPTY>
<!ELEMENT false EMPTY>
...
```

FIG. 3.1: Une partie de la définition d'AST de notre langage jouet (`tiny`) avec notre langage interne (à gauche) et son équivalence en DTD (à droite)

- atomique sans fils ou feuille (par exemple `var`)
- d'arité fixe et de types différents (`affect`)
- d'arité variable (liste) à type fixe (`statements`)

La deuxième catégorie de constructeurs permet de définir des fils optionnels, obligatoires ou de liste. Par exemple, `op(A[] aList, B? bSon, C cSon)` indique que le premier fils du constructeur `op` est une liste de `A`, le deuxième est optionnel de type `B` et le troisième est obligatoire de type `C`. Dans ce cas, les contraintes sont que les types des fils soient disjoints deux à deux pour que le système sache à quel fils se réfère le nœud courant. Il est aussi possible de déclarer des informations associées aux constructeurs sous forme d'annotations typées plus communément appelées attributs. Par exemple, la Figure 3.2 montre les attributs du constructeur `affect` utiles pour la génération d'un analyseur syntaxique et de l'afficheur associé.

```
affect(Var variable, Exp value)
  with attributes {
    fixed String separator1 = "=",
    fixed String afterOp = ";",
    fixed String styleS1 = "kw"
  }
```

```
<!ELEMENT affect (%Var;, %Exp;)>

<!ATTLIST affect
  separator1     CDATA #FIXED '='
  afterOp        CDATA #FIXED ';'
  styleS1        CDATA #FIXED 'kw'
>
```

FIG. 3.2: Définition du constructeur `affect` avec les sucres syntaxiques utiles à la génération d'un analyseur syntaxique et de l'afficheur associé ; à gauche avec notre langage interne et à droite en DTD

3.2.2 Implantation au-dessus de l'API DOM

Nous souhaitons utiliser le plus possible les composants logiciels existants issus des standards du W3C, comme par exemple l'API DOM (*Document Object Model*) de manipulation d'arbres XML. Cette API manipule des nœuds de type uniforme `org.w3c.dom.Node`. Mais l'utilisation du patron visiteur (cf. paragraphe 3.3.1 page 77) nécessite une structure fortement typée. Dans notre cas, cela signifie que le type de chaque nœud dépend du constructeur auquel il est associé. Nous avons étendu et complété cette API afin de travailler sur des arbres fortement typés. Par exemple, un nœud `affect` sera une instance de la classe `tiny.ast.AffectNodeImpl` qui étend la classe de base `org.w3c.dom.Node` comme le montre la Figure 3.3. L'avantage de construire un arbre typé est que sa cohérence est garantie par le vérificateur de types de Java. Les classes (`AffectNodeImpl, etc.`) sont automatiquement générées par SmartTools à partir de la définition d'AST (cf. Figure 3.1). Par constructeur, SmartTools génère une classe et une interface (la Figure 3.4 montre l'interface générée pour le constructeur `affect`) et une interface par type ; celle-ci est implantée par tous les constructeurs qui sont inclus dans ce type.

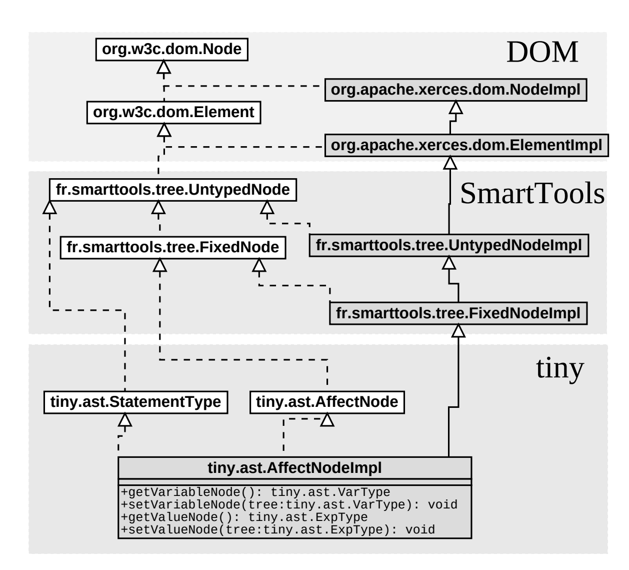


FIG. 3.3: Schéma du graphe d'héritage du constructeur `affect`.

Chaque classe Java décrivant un constructeur étend une implantation de DOM. Ces classes contiennent les méthodes d'accès (par exemple, `getVariableNode`) et de modification (`setVariableNode`) des fils et des annotations (`getSeparator1Attr`). Le nommage des fils des constructeurs (`statementList` pour le constructeur `statements`) est utilisé pour la génération des noms des accesseurs (dans ce cas `setStatementListNode` et `getStatementListNode`).

Dans la version 2 de l'API DOM, les attributs ne peuvent être que de type `String`. Comme il est parfois nécessaire lors d'un calcul de conserver des objets de type plus com-

```
package tiny.ast;

public interface AffectNode extends EVERYType, StatementType  {
   public tiny.ast.VarType getVariableNode();
   public void setVariableNode(tiny.ast.VarType node);
   public tiny.ast.ExpType getValueNode();
   public void setValueNode(exp.ast.ExpType node);

   // Attributes for affect operator
   public java.lang.String getSeparator1Attr();
   public java.lang.String getAfterOpAttr();
   public java.lang.String getStyleS1();
}
```

FIG. 3.4: Interface `AffectNode` générée

plexe dans les nœuds, nous avons ajouté la possibilité d'avoir des attributs de type autre que `String` mais ils sont volatiles. Ils n'apparaissent pas dans le format XML du document (programme) et sont perdus lors de la sauvegarde (*sérialisation*). Donc les attributs sont soit de type `String` à valeur constante, obligatoire ou optionnelle, soit de type quelconque mais volatiles (ce sont des attributs de travail seulement utiles pour des calculs sémantiques).

Cette couche au-dessus de DOM est compatible avec l'utilisation de tous les outils liés aux technologies XML comme les moteurs de transformation XSLT ou le mécanisme de références des chemins XPath (*XML Path language*; elle peut aussi utiliser les services proposés par l'API DOM dont la représentation de l'arbre au format XML.

Cette couche permet d'obtenir les informations contenues dans la définition d'AST (type du constructeur attendu, arité du constructeur, etc.), d'ajouter la notion de numéro de fils, de maintenir la cohérence de l'arbre s'il est modifié et de gérer des attributs volatiles de type complexe. Les classes des constructeurs n'héritent pas directement de DOM mais d'une des trois classes faiblement typées regroupant les informations communes à la catégorie du constructeur (feuille, liste ou variable à types différents). Avec ce typage faible, il est possible de décrire des traitements génériques[4] (par exemple, pour construire une représentation graphique de l'arbre) qui ne reposent que sur la catégorie des constructeurs.

3.2.3 Passerelles pour importer d'autres formalismes (DTD, Schema)

Il est important que les concepteurs de langages puissent définir leurs langages (définition d'AST) en utilisant directement les formats proposés par le W3C (DTD, Schema) et pas nécessairement le format propriétaire de SmartTools.

Le principal problème rencontré lors de la réalisation de l'application d'importation de DTD a été d'inférer les types nécessaires aux outils sémantiques de SmartTools. En effet, il n'existe pas explicitement de notion de type (ensemble d'éléments) dans une DTD. Avec la notion d'entité paramétrée, il est possible de définir un groupement d'éléments mais seulement à des fins de factorisation. Dans une première approche, on peut supposer que les

[4] Ces traitements ne seront pas détaillés dans cet article.

parties droites des définitions d'éléments ne soient composées que par des références à des entités paramétrées.

Par exemple, seule la première de ces deux définitions d'éléments est acceptée :

```
<!ELEMENT while ((%ConditionExp;), (%Statements;))>
<!ELEMENT while ((true|false|var|equal|notEqual), (statements))>.
```

Les éléments (<!ELEMENT while ...) sont vus comme des définitions de constructeur et leurs parties droites ne devraient être composées que de références vers des entités paramétrées (%ConditionExp;) pour indiquer le type de leurs fils. Afin de traiter le plus de DTDs possibles, il est nécessaire de définir un algorithme d'inférence de type. Par exemple pour la deuxième définition, on infère un type qui regroupe l'ensemble des éléments qui définissent une expression conditionnelle (comme l'entité %ConditionExp;).

Pour les Schema, la notion de type est explicitement présente, mais il existe des mécanismes d'extension ou de restrictions (de type) que nous devons prendre en compte lors de la traduction des Schema vers notre formalisme.

3.2.4 Outils générés

Le format XML d'un langage est essentiellement un format d'échange de données entre applications, pas vraiment adapté pour l'édition et la manipulation directes. Pour contourner ce problème, le concepteur peut définir une «vraie» syntaxe concrète à son langage (voir la Figure 3.5), donc écrire un analyseur syntaxique et l'afficheur associé. Mais cette tâche demande des compétences en techniques d'analyse syntaxique. Dans les cas simples (syntaxe non ambiguë, sans notion de priorité des constructeurs arithmétiques), cette tâche peut être automatisée. Le concepteur doit juste indiquer en supplément dans la définition d'AST les expressions régulières et les sucres syntaxiques (cf. Figure 3.2 attributs afterOp et separator1) enrobant les constructeurs. Avec ces informations (ajoutées aux constructeurs), notre outil peut produire la spécification d'un analyseur syntaxique pour le générateur ANTLR [2] composée d'une partie lexicale et d'une partie syntaxique avec les fonctions de construction d'arbre correctement typées. Il serait très facile de l'adapter à d'autres formats d'analyseur syntaxique LL(k) écrits en Java dont JavaCC [12]. Par contre, il faudrait modifier l'algorithme de génération pour d'autres méthodes d'analyse syntaxique, comme la méthode LALR du générateur CUP [8]. Notre outil génère aussi une spécification de l'afficheur associé (cf. partie 3.5.3 page 85) décrivant comment représenter les constructeurs. Cette possibilité est utilisée pour deux de nos langages internes (xprofile et Xpp présentés respectivement en 3.3.1 et 3.5.3).

La Figure 3.6 présente toutes les spécifications qui peuvent être générées à partir d'une définition d'AST :

- l'ensemble de classes et d'interfaces décrivant les constructeurs et les types (cf. 3.2.2),
- les classes de base utiles pour définir des analyses sémantiques (voir section suivante),
- un analyseur syntaxique et l'afficheur associé si des informations complémentaires sont ajoutées à la définition du langage,
- un fichier de ressources minimal qui contient des informations utiles à l'analyseur syntaxique et à l'éditeur structuré dédié au langage,
- la DTD équivalente pour valider les documents XML produits ou le Schema.

```
int table
int i
int resultat
{
   table = 2;
   i = 1;

   while (i!=10)
   {
      resultat = (i*table);
      i = (i+1);
   }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<program>
 <decls>...</decls>
 <statements>
  <affect>
   <var>table</var>
   <int>2</int>
  </affect>
  <affect>...</affect>
  <while>
   <notEqual>
    <var>i</var>
    <int>10</int>
   </notEqual>
   <statements>...</statements>
  </while>
 </statements>
</program>
```

FIG. 3.5: Programme `tiny` (table de multiplication de 2) écrit en utilisant la syntaxe concrète de `tiny` (à gauche) ou en XML (à droite).

SmartTools offre naturellement un éditeur structuré spécialisé pour chaque langage. C'est un composant générique disponible en standard pour chaque langage.

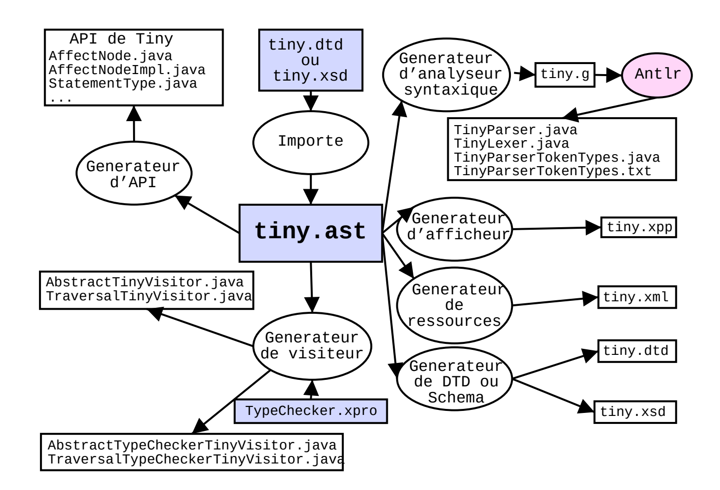


FIG. 3.6: L'ensemble des spécifications générées à partir d'une définition d'AST

3.3 Traitements sémantiques

Cette section présente les outils utiles à la conception d'applications (traitements sémantiques) sur des ASTs.

3.3.1 Le patron visiteur

Tous ces outils dérivent du patron de conception visiteur et de la programmation par aspects. Pour mieux comprendre la conception des outils, nous rappelons brièvement l'idée de base de la technique de programmation du patron visiteur.

Pour ajouter un traitement dans une hiérarchie de classes modélisant un AST (voir Figure 3.3), il suffit d'introduire une nouvelle méthode dans la classe de base. Pour définir le comportement pour un nœud de type `N`, il faut implanter cette méthode dans la classe `N`. Le traitement est donc «éclaté» dans chacune des classes représentant un nœud de l'AST. Ainsi, si `M` traitements sont définis sur un AST, chaque classe contiendra ces `M` méthodes. Il est problématique d'avoir un code réparti sur toutes les classes pour chaque traitement : maintenance de code difficile, lisibilité réduite, etc. Le patron visiteur a été introduit pour résoudre partiellement ce problème. Les méthodes communes à un traitement sont regroupées en une seule classe, dénommée visiteur qui correspond à un traitement sémantique spécifique. Cette classe contient une méthode `visit(N)` pour chaque classe `N` de l'AST. Pour mettre en œuvre cette technique, il faut que chaque classe de l'AST soit équipée d'une méthode générique `accept(visiteur)` qui délègue l'exécution à la méthode `visit(N)` appropriée dans l'objet visiteur. L'article de J. Palsberg et B. Jay [126] a servi de point de départ à nos travaux.

A partir de la définition d'AST, SmartTools génère automatiquement des fichiers Java, `AbstractVisitor` et `TraversalVisitor`, implantant une des variantes du patron visiteur. Le visiteur abstrait, `AbstractVisitor`, déclare toutes les méthodes `visit` (une par constructeur). Le visiteur de parcours, `TraversalVisitor`, hérite du visiteur abstrait en implantant toutes les méthodes `visit` de façon à effectuer un parcours en profondeur de l'arbre. Ce visiteur peut être étendu par héritage et ses méthodes `visit` surchargées pour réaliser une nouvelle analyse.

Il est aussi possible de personnaliser les signatures de ces méthodes `visit` à l'aide d'un fichier de description dénommé `xprofile`. La granularité de cette personnalisation se situe au niveau des types : il faut définir un profil pour chaque type contenu dans la définition d'AST et non pas un profil par constructeur. Pour un type donné, il sera possible de préciser le type Java de retour, le nom de la méthode et le nombre, les types Java et les noms des paramètres. L'utilisation de cette possibilité évite les défauts de la technique des visiteurs : un code illisible à cause des nombreuses coercitions de type (*casts*) en retour des méthodes `visit` ou sur les arguments (vus auparavant comme un objet de type `java.lang.Object`) et l'usage de variables globales.

La Figure 3.7 présente une partie du fichier de personnalisation de l'évaluateur de `tiny` qui parcourt l'arbre et fait évoluer les valeurs des variables. A partir de cette spécification, le système génère automatiquement les visiteurs abstraits et de parcours correctement typés et ayant les arguments voulus. On peut remarquer que les noms des méthodes `visit` sont `eval`, `evalBoolean` ou `evalInteger`, que les types de retour sont différents (par exemple de type `Boolean` pour l'évaluation d'une condition cf. ligne 15). Pour comparer, la Figure 3.9 montre la même méthode `visit` que la Figure 3.8 : il n'y a plus de coercitions de type en `Boolean` en ligne 2 et l'argument `env` est correctement typé.

Le langage `xprofile` permet également de préciser le parcours à effectuer dans l'arbre

```
XProfile EvalTinyVisitor;
Formalism tiny;
  import tiny.visitors.TinyEnv;
  import java.lang.Boolean;
  import java.lang.Integer;

Profiles
  Object eval(%Top, TinyEnv env);
  Object eval(%Decls, TinyEnv env);
  Object eval(%Decl, TinyEnv env);
  Object eval(%Statements, TinyEnv env);
  Object eval(%Statement, TinyEnv env);
  Object eval(%Exp, TinyEnv env);
  Integer evalInteger(%ArithmeticExp, TinyEnv env);
  Boolean evalBoolean(%ConditionExp, TinyEnv env);
  Boolean evalBoolean(%ConditionOp, TinyEnv env);
  Object eval(%ArithmeticOp, TinyEnv env);
  Object eval(%Var, TinyEnv env);
  ...
```

FIG. 3.7: Partie du fichier de personnalisation de l'évaluateur de `tiny`

```
public Object visit(WhileNode node, Object env) throws VisitorException {
   Boolean cond = (Boolean)visit(node.getCondNode(),env);

   if (cond.booleanValue()) {//tantque condition vérifiée
      visit(node.getStatementsNode(), env); //execute le bloc
      visit(node, env); //ré-exécute la visite sur ce noeud -> récursion
   }
   return null;
}
```

FIG. 3.8: Evaluation du constructeur `while` sans profil

```
public Object eval(WhileNode node, TinyEnv env) throws VisitorException {
   Boolean cond = evalBoolean(node.getCondNode(),env);

   if (cond.booleanValue()) {//tantque condition vérifiée
      eval(node.getStatementsNode(), env); //execute le bloc
      eval(node, env); //ré-exécute la visite sur ce noeud -> récursion
   }
   return null;
}
```

FIG. 3.9: Evaluation du constructeur `while` avec un profil

(du nœud de départ vers les nœuds d'arrivées) pour un visiteur. De cette façon, seuls les nœuds présents sur les chemins choisis sont visités. Cela permet de réduire de façon significative le temps d'exécution des visiteurs. Une analyse de dépendance de graphe sur la définition d'AST est effectuée pour générer les visiteurs correspondants à ce parcours.

L'introduction des signatures des méthodes interdit l'utilisation de la variante de base

des visiteurs (utilisant un appel explicite à une méthode `accept`); sinon il faudrait avoir une méthode `accept` par signature dans les classes des nœuds. Il est donc nécessaire d'utiliser la réflexivité de Java pour trouver la méthode `visit` à appeler en fonction du type du nœud et des arguments. Ce problème est bien connu, surtout dans le cadre de Java et correspond aux travaux de recherche sur les multi-méthodes [120]. Nous utilisons donc une variante des visiteurs basée sur l'introspection de Java. Une méthode générique (appelée `invokeVisit`) est exécutée à chaque appel d'une méthode `visit` pour trouver la méthode à appeler. Nous avons utilisé dans un premier temps une implantation des multi-méthodes pour Java [70] qui correspondait à notre problème. Cependant, l'ensemble des méthodes `visit` (ou signatures) est connu d'avance (ensemble borné) dans notre cadre. L'implantation du mécanisme d'invocation des méthodes a donc été remplacée par une version plus simple utilisant une table d'indirection pour rechercher la méthode `visit` adéquate. Cette table, pré-calculée lors de la génération des visiteurs, indique pour chaque couple `(type, constructeur)` la référence de la méthode à appeler.

En fait, notre approche est une spécialisation de celle des multi-méthodes. Nous avons comparé les deux approches (multi-méthodes et génération d'une table) et les performances sont équivalentes en temps d'exécution. L'intérêt de l'approche par génération d'une table d'indirection, outre sa simplicité de mise en œuvre, est de permettre d'associer d'autres traitements comme l'introduction d'aspects à chaque appel d'une méthode `visit`.

3.3.2 Programmation par aspects

Il a suffi de modifier légèrement la méthode `invokeVisit` pour exécuter du code avant et après les appels effectifs aux méthodes `visit`. Ainsi, on obtient une programmation par aspects [101] spécifique à nos visiteurs sans transformation de programme, contrairement aux premiers outils d'AOP (*Aspect-Oriented Programming*) [4]. Nos points de jonction sont limités à avant et après une méthode `visit`. On peut définir ces aspects sur un constructeur, sur un type de nœud ou sur tous les nœuds. La Figure 3.10 présente le code d'un aspect permettant de tracer toutes les méthodes `visit` appelées. Plusieurs aspects différents peuvent être branchés sur un même visiteur. Ils seront alors exécutés séquentiellement dans l'ordre de branchement. Ce branchement (mais aussi le débranchement) peut se faire dynamiquement et à tout moment pendant l'exécution d'un visiteur. On peut donc modifier dynamiquement le comportement d'un visiteur par ajout ou retrait d'aspects.

Par exemple, cette technique est employée pour fournir un mode générique d'exécution pas-à-pas graphique (dit *mode debug*) à nos visiteurs. Il suffit de brancher sur chaque appel de méthode `visit` un aspect standard qui gère la communication entre la fenêtre de dialogue (fenêtre de *debug*) et l'utilisateur.

L'utilisation conjointe des visiteurs et des aspects fournit une technique simple et puissante de développement d'outils d'analyses dédiés à un langage. Actuellement, nous travaillons sur une extension de cette technique pour découpler le parcours des traitements (actions sémantiques). L'idée est de pouvoir spécifier les actions indépendamment d'un parcours. Ainsi, au lieu de coder le parcours dans les méthodes `visit`, on construit un objet sachant effectuer ce parcours et les actions ne sont décrites qu'avec des aspects. L'intérêt de cette extension est de permettre la conception de traitements par composition d'aspects.

```
package fr.smarttools.debug;
import fr.smarttools.vtp.visitorpattern.Aspect;
import fr.smarttools.vtp.Type;

public class TraceAspect implements Aspect {
   public void before(Type t, Object[] param) {
      // param[0] est le noeud courant
      System.out.println ("Debut visit sur " + param[0].getClass());
   }
   public void after(Type t, Object[] param) {
      System.out.println ("Fin visit sur " + param[0].getClass());
   }
}
```

FIG. 3.10: Code d’un aspect traçant les méthodes `visit` appelées

Avec cette extension, le vérificateur de types de `tiny` a été décomposé en deux aspects : l’un pour l’analyse de nom et l’autre pour la vérification.

3.4 L’architecture de SmartTools

Cette section présente brièvement l’architecture modulaire de SmartTools. La motivation principale de cette architecture est de construire un outil avec des composants logiciels ayant des fonctionnalités bien spécifiques comme le modèle “modèle-vue-contrôleur” de Smalltalk. Les communications (échanges de messages) entre ces divers composants sont majoritairement de type asynchrone : l’émetteur ne reste pas bloqué en attente du résultat du récepteur. Comme le nombre de composants peut devenir important, un mécanisme d’aiguillage des événements (contrôleur de messages) est nécessaire pour gérer l’ensemble des communications.

SmartTools est donc composé d’un ensemble de composants qui échangent des messages (événements typés) de manière asynchrone à travers le contrôleur de messages. Le comportement d’un composant est défini par l’ensemble des types de messages qu’il peut recevoir et émettre. Un composant doit, tout d’abord, s’enregistrer auprès du contrôleur de messages et indiquer les types de messages qu’il pourra traiter.

Le contrôleur de messages a la responsabilité de gérer le flux de messages et de les aiguiller pour les délivrer à leur(s) destinataire(s). Il s’agit d’un bus logiciel spécifiquement développé pour les besoins de SmartTools.

Les principaux composants sont brièvement décrits ci-dessous :

– **Document**
  Chaque document contient un AST. Dans la Figure 3.11, `Document 1` et `Document 2` correspondent à des ASTs sur lesquels un utilisateur travaille. `Document IG` joue un rôle particulier : c’est l’AST correspondant à la structure de l’interface graphique de SmartTools.
– **Vue**
  Chaque vue est un composant indépendant qui montre le contenu d’un document se-

lon le type d'affichage. Par exemple, certaines vues vont afficher l'AST sous forme textuelle avec une syntaxe colorée, d'autres vont en donner une représentation graphique.

– **Interface utilisateur**
  Le module d'interface utilisateur a pour rôle de créer des vues et de gérer les différents menus et la barre d'outils.
– **Les gestionnaires d'analyseurs syntaxiques et de documents**
  Le premier composant choisit l'analyseur syntaxique approprié en fonction de l'extension du fichier reçu. Puis il exécute cet analyseur pour produire l'AST. Le gestionnaire de documents construit des composants document à partir d'ASTs reçus et les connecte au contrôleur de messages.
– **Base**
  La base est un composant qui contient toutes les définitions de ressources utilisées par SmartTools : définitions de styles, de menus, de couleurs, de fontes, etc. Ces définitions sont stockées dans la base sous forme d'ASTs.

Pour fixer les idées, la Figure 3.11 montre une configuration possible de ces divers types de composants.

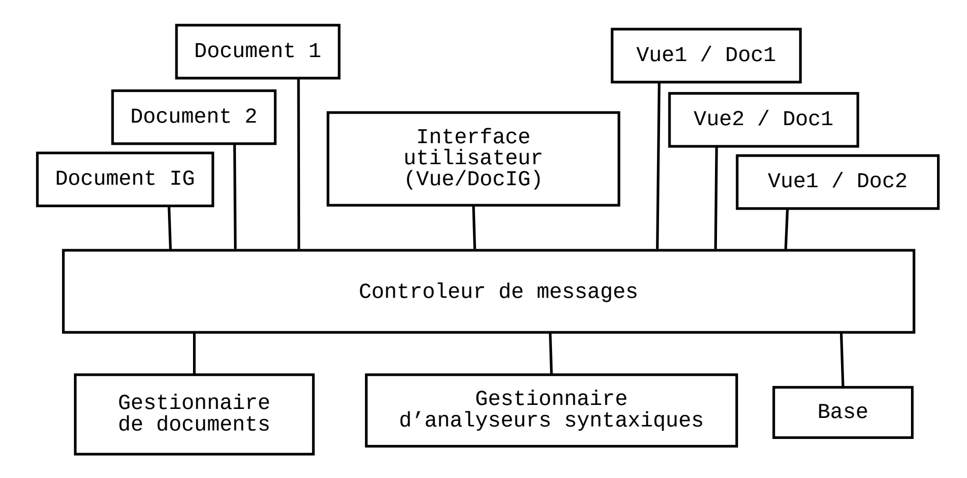


FIG. 3.11: L'architecture de SmartTools

Il est important de préciser que la communication entre composants passe obligatoirement par l'échange de messages à travers notre bus logiciel. Cette contrainte est un moyen simple pour imposer que chaque composant soit «proprement écrit». Ainsi un composant n'a pas de référence directe sur des objets appartenant à d'autres composants. Avec cette discipline de programmation, les composants peuvent être facilement exportés ou importés.

L'ensemble des types de messages est extensible. C'est l'une des techniques proposées pour étendre les fonctionnalités de SmartTools afin de répondre à des besoins spécifiques ou pour intégrer SmartTools dans des environnements de travail déjà existants.

Les messages sont constitués de deux parties : une partie concernant les informations nécessaires à l'acheminement du message et une partie concernant les données transportées. La première partie est commune à tous les types de messages et est donc gérée par une

classe abstraite dont ils héritent. La deuxième partie est spécifique à chaque type de message et contient les données utiles. Les données les plus couramment échangées entre ces composants sont des arbres. Le format XML comme protocole d'arbre est naturellement le bon support pour ces échanges. Ce format possède l'avantage d'être proche de la structure des messages SOAP. En effet, il est possible de placer les informations liées à l'acheminement du message (type d'action, identifiant du module expéditeur et éventuellement identifiant du module destinataire) dans l'entête (balise `HEADER`) d'un message SOAP. Les données utiles peuvent être placées dans le corps du message (balise `BODY`). C'est ainsi que le contrôleur de messages de SmartTools a été doté de filtres capables d'importer ou d'exporter des messages en respectant les spécifications SOAP. Cela offre à SmartTools la capacité d'échanger des messages à travers un réseau avec des modules qui ne sont pas forcément écrits en Java mais qui veulent bénéficier de certaines fonctionnalités de SmartTools.

3.5 Environnement interactif

Cette section présente les concepts et mécanismes qui ont permis la réalisation d'une interface graphique conviviale, exportable et aisément configurable pour la plate-forme SmartTools.

3.5.1 Modèle document/vues

L'interface graphique est basée sur le modèle document/vues qui s'intègre particulièrement bien dans l'architecture de SmartTools. Le composant document s'occupe des traitements (visiteurs, persistance, etc) sur un AST et les composants vues des représentations graphiques ou textuelles de cet AST. Ce modèle nous apporte une bonne séparation des fonctionnalités et la possibilité d'avoir plusieurs types de vues sur un même document (voir Figure 3.12). Il est conçu de manière à conserver en permanence l'isomorphisme entre les vues et le document.

Comme tous les composants de SmartTools, un document et ses vues s'échangent des messages dont le contenu est codé dans un format XML. Cela implique qu'il n'y ait pas de référence directe entre les nœuds de l'AST et les objets graphiques de ses vues. L'isomorphisme est alors garanti par échanges de messages (voir Figure 3.13) contenant trois types d'informations : l'action, son emplacement et éventuellement le sous-arbre modifié. L'action est exprimée par le type du message, l'emplacement par un chemin absolu sous forme de XPath et le sous-arbre par sa représentation en XML.

3.5.2 Construction des vues et de l'interface graphique

Les vues sont construites en appliquant une transformation à la représentation XML d'un arbre, puis en interprétant son résultat avec un afficheur approprié.

Cette transformation peut être effectuée soit à l'aide d'un visiteur après avoir reconstruit un arbre à partir des données XML, soit par l'utilisation d'un moteur XSLT. C'est cette deuxième technique (voir Figure 3.14) qui a été retenue dans SmartTools pour plusieurs raisons :

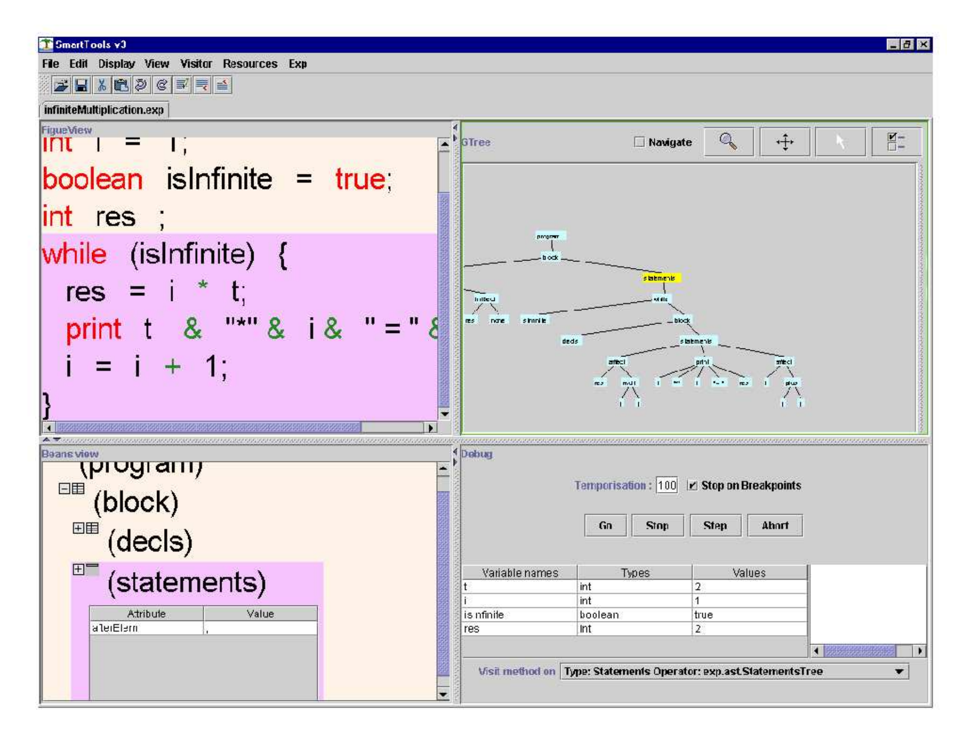


FIG. 3.12: Exemple d'interface utilisateur composée de quatre vues différentes du même AST : la vue textuelle (syntaxe concrète) située en haut à gauche, la vue sous forme d'arbre graphique à droite, la vue sous forme de menus et d'attributs en bas à gauche, et enfin la vue du *mode debug* d'un visiteur d'évaluation.

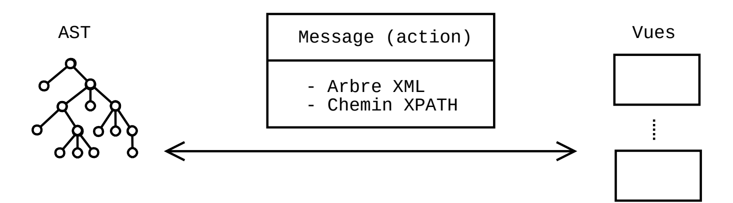


FIG. 3.13: Communication entre le document et ses vues

– elle évite d'avoir une copie du document côté vue ;
– elle n'oblige pas à utiliser Java pour la transformation et l'affichage ;
– elle permet l'envoi des transformations à travers le réseau pour déplacer la construction de vues côté client ;
– il s'agit d'une recommandation du W3C et non d'une technique propriétaire.

Le paragraphe 3.6.1 montre comment cette technologie facilite l'exportation de vues vers un navigateur Web.

La mise en page des vues et l'organisation de l'interface graphique sont décrites par un AST sur lequel on peut appliquer les concepts de création des vues. L'interface graphique est donc une vue particulière de cet AST. Ce dernier peut aisément être rendu persistant, et

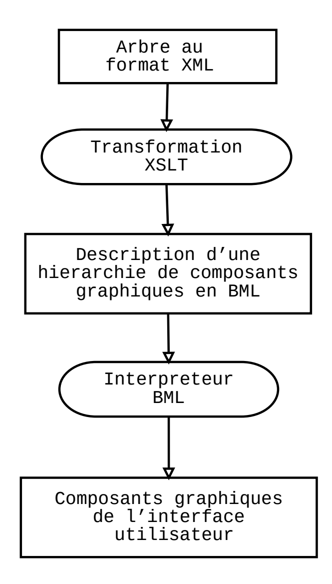


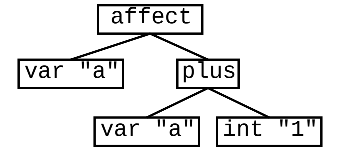
FIG. 3.14: Processus de transformation.

l'interface devient facilement configurable.

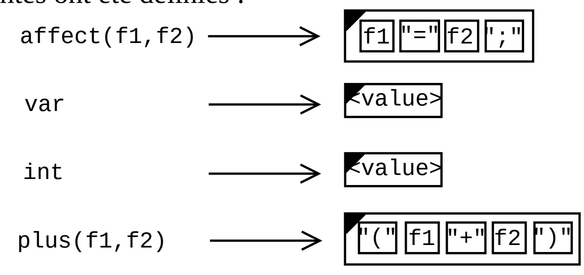
Les composants graphiques utilisés pour la version Java des vues sont basés sur la bibliothèque `javax.Swing`. Le moteur XSLT ne produisant que du format texte, il faut disposer d'un format de description des objets et de l'interpréteur associé pour les construire. Le format BML (*Bean Markup Language*) [82] répond parfaitement à ce besoin.

Nous allons détailler la procédure de transformation et la technique de marquage des nœuds qui permettent de maintenir les vues isomorphes à l'AST.

Prenons en exemple l'expression `a=a+1` correspondant à l'AST :



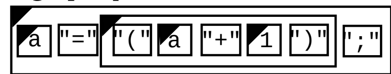
Pour obtenir une vue textuelle (syntaxe concrète) de cet arbre, les transformations suivantes ont été définies :



Chaque règle de transformation indique comment transformer un constructeur en une hiérarchie de composants graphiques. Les composants graphiques correspondants à des nœuds de l'AST sont marqués pour les différencier des sucres syntaxiques (`=`, `+`, etc). Cette technique permet de calculer la correspondance entre les composants graphiques et

les nœuds de l'AST : seuls les composants marqués sont pris en compte lors du calcul du chemin.

Dans notre exemple, la transformation de l'arbre va produire la hiérarchie de composants graphiques suivante :



La vue générique (cf. la Figure 3.12 à gauche en bas) a été réalisée en créant une nouvelle fonctionnalité de formatage qui hérite d'un seul composant `Swing`. Cette vue montre le contenu d'un AST, de manière interactive, avec les attributs de chaque nœud dans des tables que l'on peut cacher ou afficher à volonté. Les calculs d'alignement et de mise en page sont effectués par les composants `Swing`. La création et l'intégration de nouveaux composants sont ainsi simplifiés.

3.5.3 Le langage Xpp

Cette procédure d'affichage présente cependant plusieurs inconvénients. Tout d'abord, BML et XSLT sont des langages XML peu lisibles et très redondants. Ensuite, XSLT autorise des transformations d'arbres ascendantes (sur les ancêtres du nœud sélectionné) et descendantes (sur les sous-arbres). Dans SmartTools, seules les transformations sur les sous-arbres doivent être autorisées pour conserver un calcul de chemin cohérent. Ainsi, si un nœud ou l'un de ses descendants est modifié lors de l'édition structurée, seul le réaffichage de ce nœud est nécessaire et non celui de l'arbre entier. Ces contraintes nous ont amenés à définir un langage de filtrage (*pattern-matching*) appelé Xpp, traduit ensuite en XSLT. Ses fonctionnalités sont proches de celles de XSLT mais sa syntaxe est beaucoup plus simple et concise (voir Figures 3.15 et 3.16). Le langage Xpp est un préprocesseur à XSLT. De plus, dans Xpp, seules les transformations descendantes sont autorisées, ce qui respecte ainsi la contrainte d'incrémentalité.

Les fichiers Xpp consistent en un ensemble de règles de transformations et de fonctions. Les règles de transformation (voir Figure 3.15) sont constituées de deux parties :

- en partie gauche, le filtre décrivant le nom du nœud recherché avec éventuellement des conditions sur ses fils ou attributs ;
- en partie droite, le code décrivant le formatage et les sucres syntaxiques souhaités.

```
Rules

affect(x, y) -> h(x, label("="), y, label(";"));
```

FIG. 3.15: Exemple de règle Xpp.

Ces règles permettent d'identifier les nœuds d'un arbre correspondant à un certain motif, et de définir pour ces nœuds une mise en forme générale (alignement horizontal ou vertical, indentation, espacement, etc) indépendante du format de sortie (BML, HTML ou texte), comme le montre la Figure 3.17.

```
<xsl:template match="affect[*[1]][*[2]][count(*)=2]">
  <xsl:variable name="x" select="./*[1]"/>
  <xsl:variable name="y" select="./*[2]"/>
  <bean class="fr.smarttools.view.GNodeContainer">
     <add>
       <xsl:apply-templates select="$x"/>
     </add>
     <add>
       <bean class="fr.smarttools.view.FJLabel">
         <args>
           <string>=</string>
         </args>
       </bean>
     </add>
     <add>
       <xsl:apply-templates select="$y"/>
     </add>
     <add>
       <bean class="fr.smarttools.view.FJLabel">
         <args>
           <string>;</string>
         </args>
       </bean>
     </add>
  </bean>
</xsl:template>
```

FIG. 3.16: Règle de la Figure 3.15 exprimée en XSLT.

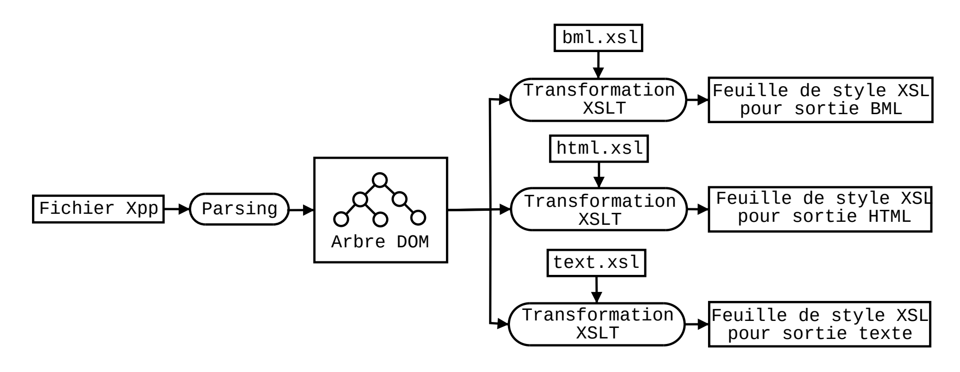


FIG. 3.17: Processus de génération des feuilles de style décrites par le fichier Xpp pour chaque format de sortie.

Chaque format de sortie possède une feuille de style définissant les fonctions de formatage (`label`, `h`, etc). Le processus de transformation s'effectue en deux étapes :

– La première étape transforme le fichier Xpp du langage avec la feuille de style du format de sortie souhaité ;
– Puis, la feuille de style obtenue est appliquée au document.

La feuille de style de chaque format de sortie définit la traduction de fonctions de mise en forme ; par exemple, la fonction d'alignement horizontal, `h`, en BML, HTML ou texte. Il est possible d'étendre Xpp en ajoutant de nouvelles fonctions de mise en forme. Il suffit

de définir le résultat de ces fonctions dans les trois fichiers XSLT, pour pouvoir les utiliser ensuite dans Xpp.

Le but est de mettre à disposition des utilisateurs un ensemble de boîtes d'affichage (horizontales, verticales). Ceux-ci n'ont plus qu'à décrire l'affichage sous forme de règles pour chaque opérateur de leur langage sans se soucier de l'implantation des boîtes. L'utilisateur garde cependant la possibilité de réécrire ou d'ajouter des fonctions si l'ensemble des fonctions prédéfinies ne lui suffit pas.

3.6 Applications

Cette section présente une partie des applications réalisées grâce à SmartTools. Il convient tout d'abord de noter que tous ses langages internes ont été construits et définis en l'utilisant. Nos techniques de programmation par visiteur et afficheur sont largement utilisées au sein de SmartTools, à tous les niveaux : environ 40% de son code source est automatiquement généré.

3.6.1 Une application d'interconnexion avec un afficheur Web

Grâce à l'architecture modulaire et à l'utilisation des technologies XML, toute vue de SmartTools peut être envoyée sur un navigateur Web, à travers le protocole HTTP. L'idée de base consiste à installer un mécanisme clients/serveur par type de vue. Côté client, l'interpréteur BML construit les objets graphiques. Côté serveur, la transformation est appliquée pour produire une description BML d'une vue. Nous avons expérimenté ces concepts par l'utilisation d'*applets* pour les clients et de *servlet* connectée au bus SmartTools pour le serveur (voir Figure 3.18). On peut généraliser cette approche en utilisant les *Web Services* et le protocole SOAP. Ainsi, nous avons présenté la sélection d'un nœud dans l'AST sous la forme d'un *Web Service*. Puis dans la plate-forme .NET, nous avons écrit un client .NET en C# qui fait appel à ce service de sélection par le protocole SOAP [162]. Cette expérience nous a montré qu'il était facile de connecter SmartTools à des environnements hétérogènes.

3.6.2 Environnement dédiés

Accepter le format DTD en entrée de notre outil présente l'intérêt de pouvoir produire un environnement minimal (voir Figure 3.6, page 76) pour n'importe quel langage défini avec une DTD. L'exemple type a été la réalisation d'un environnement pour l'outil Ant [3] qui est un «système de `make`» écrit en Java dont les règles de dépendances sont exprimées en XML. L'utilisation de la DTD de Ant a permis de construire cet environnement avec un coût de développement quasiment nul.

Un autre outil a été développé pour la visualisation de formules mathématiques en utilisant la norme SVG (*Scalable Vector Graphics*) qui offre un rendu graphique de très bonne qualité. Une conversion de MathML 2.0 (*Mathematical Markup Language*) vers SVG a été réalisée avec des visiteurs et l'affichage proprement dit effectué grâce à l'intégration d'une vue de *Batik* [5] dans SmartTools.

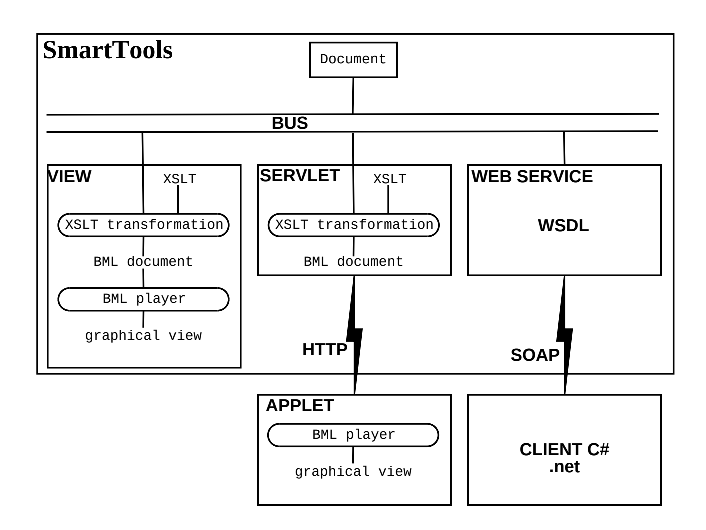


FIG. 3.18: Différent types d'accès à SmartTools

Un environnement plus complexe (pour Java Card, langage de programmation des cartes à puce) a aussi été élaboré dans le cadre d'un contrat industriel avec CP8 Schumberger et a montré que le passage à l'échelle était possible avec SmartTools.

3.7 Discussion et Perspectives

Dans cette section, en comparant notre approche à d'autres travaux, nous allons, sur chaque point important de notre démarche, motiver nos choix, donner les perspectives futures et souligner les principales innovations. Comme notre outil supporte et utilise différentes familles technologiques, une comparaison exhaustive et complète s'avère une tâche difficile et fastidieuse. Il nous a semblé beaucoup plus instructif de préciser pour chaque famille quels sont les problèmes ou les verrous technologiques connus. A partir de ces indications, nous expliquerons plus particulièrement en quoi notre démarche nous semble être la plus appropriée pour résoudre ces problèmes ; puis pourquoi nos choix permettent de mieux s'adapter aux futures évolutions ou sauts technologiques.

Édition Structurée et environnement interactif

Dans le domaine des éditeurs structurés [140, 31, 102], les langages métiers définis à l'aide des formalismes XML sont certainement de meilleurs candidats que les langages dits de programmation, où les éditeurs professionnels et spécialisés (*Visual Studio*™, *JBuilder*™, *Visual Age*™, etc) sont des concurrents manifestes à l'approche générique. Mais il est important, même pour ces langages métiers [161] beaucoup moins exigeants en terme d'édition libre (possibilité d'écrire du code à la volée), de proposer des outils d'affichage ouverts et extensibles à de nouvelles bibliothèques de composants graphiques. Nous avons montré qu'il est relativement simple de construire au-dessus de l'outil de transformation XSLT, un mécanisme d'affichage de vues avec les contraintes particulières liées à l'édition

structurée. Nous montrons aussi que cette approche (avec BML) permet une exportation aisée des vues graphiques à travers le réseau. Cela montre encore une fois tout l'avantage d'utiliser diverses familles technologiques : la bibliothèque `Swing` pour le graphisme, l'outil XSLT pour les transformations, XML et BML pour la *sérialisation*. Nous n'avons pas encore exploité toutes les possibilités de nos outils d'affichage. L'utilisation des technologies XML est un atout indéniable de notre approche.

Passerelle vers les formalismes DTD et *Schema*

Nos efforts pour accepter les formalismes du W3C sont certes motivés par notre souci d'élargir le champ d'applications de SmartTools. L'intérêt est de proposer pour ce type d'applications (langages) nos outils d'édition, d'affichage et/ou de description sémantique. Notre approche de génération automatique du couple analyseur syntaxique et afficheur semble envisageable pour des langages métiers simples. Elle serait certainement trop complexe pour des langages de programmation. Cette génération devrait rendre de grands services dans ce contexte de petits langages métiers. Les formalismes du W3C avaient comme vocation initiale de mieux structurer les informations issues des documents. Aujourd'hui, ils sont aussi utilisés pour la manipulation de données de natures différentes. Par exemple, le format XMI (*XML Metadata Interchange*) [19] est utilisé pour la représentation des diagrammes UML (*Unified Modeling Language*) [18], formalisme très proche des langages de programmation. Nous sommes convaincus qu'il y aura un besoin important d'outils puissants pour décrire des traitements complexes pour ces langages métiers. Il serait dommage de réinventer des outils de manipulation dans ce nouveau contexte alors qu'il existe déjà, certes dans d'autres communautés scientifiques, des travaux de recherche et des développements parfaitement adaptés à cela.

Outils sémantiques

Le succès grandissant de la notion de patrons de conception [73] montre que les concepts de programmation par objets ne sont pas suffisants et que chaque type d'application ou problématique demande des solutions appropriées [122]. En particulier, le patron visiteur a suscité un ensemble de travaux de recherche [127], qui ont tous comme objectif de trouver le meilleur compromis entre la lisibilité et l'efficacité. L'un des autres soucis de ce patron est la composition de visiteurs [104].

Dans [46], nous avions déjà remarqué certaines similitudes sur cette problématique pour des familles de technologies différentes (grammaires attribuées [145, 80], programmation polytypique [83] et programmation adaptative [128, 110]). La programmation adaptative suit cette même problématique de séparation des concepts (parcours et sémantique). La programmation par aspects [98] a aussi été introduite pour la séparation des parties fonctionnelle et non-fonctionnelle (applicative ou de services) d'une application. Les approches par transformation de programme utilisées pour la programmation par aspects ont montré leurs limites [33]. Il est clair qu'il existe des liens très forts avec les travaux de recherche sur la réflexivité [112, 113] pour les langages à objets, en particulier la notion de MOP (*Meta-Object Protocole*) [99]. Les mécanismes mis en jeu dans ces approches totalement

dynamiques ne sont pas simples d'utilisation. Ils demandent de comprendre la sémantique sous-jacente des langages à objet.

L'originalité de notre approche est de partir d'une spécification déclarative de la structure des objets et d'effectuer une génération de code source enrichie par les mécanismes de programmation par aspects ou adaptative. L'intérêt est d'une part d'éviter les problèmes d'efficacité par une génération de code source et de cacher à l'utilisateur la complexité des mécanismes mis en jeu. De plus, notre approche de programmation par aspects a le mérite d'être mise en œuvre très simplement par une extension naturelle du patron de conception visiteur.

Architecture modulaire

Pour l'architecture de notre logiciel, nous avons volontairement pris la solution de centraliser les problèmes de communication et de connexion au sein d'un bus logiciel [139, 28]. La technologie XML joue pleinement son rôle en terme de protocole de communication des données complexes. Mais cette démarche a des limites car les composants perdent la possibilité d'être intégrés dans d'autres environnements sans être forcément rattachés au bus. En effet, le bus est le seul moyen pour les composants de communiquer vers l'extérieur. Par exemple, dans [28], les auteurs ont poussé cette voie à son paroxysme, puisqu'ils spécifient la sémantique des communications à l'aide d'un langage de coordination au sein même de l'objet bus. A l'inverse, l'approche qui consiste à enrichir chaque composant par de nouveaux services se heurte essentiellement aux trois problématiques suivantes :

- avoir des composants ouverts à de nouveaux services non prévus initialement (au moment de la phase de conception) ;
- maîtriser l'ensemble des composants ce qui impose naturellement la construction d'un référentiel de l'état du système ;
- prendre en compte les diverses technologies de composant (CORBA, EJB, COM, Web Services).

Actuellement nous étudions plusieurs voies possibles pour faire évoluer notre architecture dans le but de rendre les composants indépendants du bus logiciel. Dans un premier temps, nous voulons que l'interface de communication des composants soit indépendante du choix de la technologie de composant effectivement choisie. Pour cela, la communication sera déléguée à un objet qui prendra en compte le choix de la technologie de composant effectivement utilisée. A partir d'une description de haut niveau des interfaces des composants et de la technologie de composant choisie, SmartTools générera l'implantation de cet objet de communication.

Applications

Bien que notre outil soit en cours de développement, il présente de nombreuses perspectives d'utilisation dans des domaines d'application très variés. Ces perspectives peuvent être regroupées en six catégories :

- les langages métiers de SmartTools :
  L'outil utilise les technologies objets et XML pour la description de ses langages

(définition d'AST, Xpp, xprofile) et composants, pour ses fichiers de configuration (scripts de lancement), et pour le paramétrage des vues graphiques. Cette utilisation intensive valide nos choix et est susceptible d'engendrer de nouveaux champs d'application. Par exemple, nous avons découvert que l'interface graphique pouvait être traitée comme un AST.

– les langages de programmation ou métiers :
  Le champ d'application traditionnel est la conception d'environnements pour des langages de programmation ou métiers. Par exemple un environnement pour Java est en cours de réalisation dans le but de réaliser des analyses statiques complexes utilisant les techniques de programmation par visiteurs.
– les langages de méta-modélisation :
  Depuis l'émergence d'outils de méta-modélisation autour d'UML (MOF - *Meta-Object Facility*, OCL - *Object Constraint Language*), des analogies entre ces méta-langages et les langages de programmation commencent à être identifiées. L'établissement de passerelles sont déjà à l'étude entre l'AGL de Softeam (*Objecteering*™) et SmartTools. Nos techniques de programmation par aspect pourraient être utilisées pour la description de la sémantique de ces modèles (*Action Semantic* d'UML).
– les langages du W3C :
  Pour la réalisation des passerelles entre les formalismes du W3C et nos langages de spécification, des environnements de programmation pour les DTD et les Schema ont été développés. Dans le cadre d'un contrat industriel, un environnement de transformation pour le format XHTML va être réalisé. Tous ces exemples montrent que les formalismes du W3C sont potentiellement des champs d'application pour SmartTools. Son approche générique est un atout indéniable pour la conception rapide d'environnements.
– les langages de description de composant :
  Les équipes de recherche sur le thème des composants (en particulier les travaux autour de la nouvelle norme CORBA (CCM - *CORBA Component Model*) ou les plates-formes comme *ObjectWeb*) sont des utilisateurs potentiels de SmartTools. Des environnements de développement peuvent être générés pour leurs langages métiers (par exemple le langage IDL - *Interface Definition Language*) et des générateurs (compilateurs) spécifiés à l'aide de nos techniques. Pour la description de nos composants, nous pensons utiliser le format WSDL (*Web Service Definition Language*) avec des générateurs spécifiques selon la technologie de composant choisie (CORBA, Web Services, EJB).
– les systèmes d'ingénierie ontologique (RDF - *Resource Description Framework*) du Web Sémantique :
  Le thème de recherche du Web Sémantique regroupe tout un ensemble de concepts très variés dont le lien avec SmartTools semble être l'analogie entre les systèmes d'ingénierie ontologique [56] et les notions de syntaxe abstraite (système de type). Même s'il est trop tôt pour parler d'application spécifique pour le Web Sémantique avec SmartTools, il nous semble raisonnable de penser que nos techniques (visiteur ou programmation par aspects) pourront être utilisées pour certaines applications de ce domaine.

Conclusion

Le souci majeur du développement est certainement la réutilisation au sens large du terme. Nous espérons que le lecteur a été convaincu que notre outil, par sa conception, est un exemple type de logiciel centré sur l'idée de la réutilisation grâce à son approche générique. L'utilisation de standards tant en terme de composants logiciels ou bibliothèques (le succès de Java vient principalement de la richesse des bibliothèques et des outils proposés) qu'en terme de spécifications (les formalismes du W3C) est la garantie d'une facilité d'évolution. L'une des caractéristiques importantes de notre approche est que l'outil s'appuie sur une description abstraite des objets pour générer automatiquement une grande partie des applications, aussi bien pour l'interface utilisateur, les afficheurs, les composants sémantiques et bientôt pour les interfaces de communications. L'un des grands avantages de cette phase de génération est de faciliter la prise en compte des futures évolutions des technologies et des fonctionnalités non prévues initialement.

Finalement, notre outil est notre propre champ d'expérimentation des concepts introduits. La richesse et la variété des techniques, et le champ des applications susceptibles d'être traitées démontrent l'intérêt de notre approche générique. Cette approche par génération à partir de modèles est conforme à la nouvelle stratégie de conception [37], *Model Driven Architecture* (MDA), définie par l'OMG (*Object Management Group*). En effet, les phases de génération de code source à partir de modèles décrits avec les technologies XML sont un moyen de garantir une évolution aisée de l'outil SmartTools et des environnements produits.

Chapitre 4

Modèle à composant pour SMARTTOOLS

4.1 Introduction

Avec l'émergence d'Internet, la conception et le développement d'applications complexes doivent impérativement prendre en compte les aspects de répartition (application distribuée), déploiement et réutilisation de code. Cette évolution a pour conséquence un changement radical dans la programmation de telles applications. Avec ces bouleversements, l'approche objet, adoptée pour la production de logiciels, a présenté des limitations. Les mécanismes offerts étaient de trop bas niveau pour exprimer des interconnexions complexes entre objets ou inexistants pour gérer des objets distribués sur plusieurs machines. Ces limitations ont conduit à l'émergence d'un nouveau paradigme de programmation, le composant [116, 117, 35, 13, 36], afin de mieux séparer les aspects de communication et les parties fonctionnelles, de s'abstraire des langages de programmation, et de construire des applications par assemblage de composants. Ce passage de la programmation à la petite échelle (*in the small*) à la programmation à grande échelle (*in the large*) permet une meilleure réutilisation du code.

Cet article va présenter notre démarche de conception d'une architecture par composants pour un atelier logiciel nommé SMARTTOOLS [1] [132]. Tout d'abord, les caractéristiques spécifiques à un tel logiciel sont explicitées ainsi que les contraintes ayant in-

[1]http ://www-sop.inria.fr/oasis/SmartTools

fluencé notre démarche de conception de l'architecture. L'énumération de ces contraintes nous semble importante pour justifier nos choix de conception du modèle de composants sous-jacent. De plus, nous pensons que ces contraintes, dans quelques années, seront communes à de nombreux développements logiciels et non seulement à notre application, afin d'assurer une meilleure évolution et adaptabilité aux applications produites. L'apport de cet article est donc de décrire une démarche de conception d'architecture logicielle transposable pour de nombreuses autres applications.

Avoir une architecture par composants, pour notre application, était nécessaire et ce pour plusieurs raisons :

– SMARTTOOLS est un générateur d'outils pour des langages métiers ou de programmation. Il était donc vital de pouvoir *séparer*, de manière modulaire, les outils de base génériques (le cœur du système) et ceux générés spécifiques à un langage donné afin d'éviter d'augmenter, à l'infini, la taille de notre atelier logiciel.
– Notre méta-application devait pouvoir être configurée (restreinte) en fonction de l'application souhaitée par l'utilisateur, afin d'éviter de charger en mémoire des outils inutiles ; une application étant composée par un ensemble d'outils. Il s'avère aussi important d'offrir la possibilité de charger en cours d'exécution, à la demande, de nouveaux outils associés à des langages. Son interface graphique et son architecture doivent s'adapter aux diverses configurations possibles, différentes pour chaque langage traité. Elles doivent être configurables et extensibles. Il est donc important que les applications générées soient seulement composées des *outils nécessaires* et aient des *architectures dynamiques*. Un mécanisme de gestionnaire de composants est utile pour interpréter cette configuration de composants au lancement d'une application et pour intégrer de nouveaux composants pendant l'exécution ; c'est une sorte de «machine virtuelle de composants».
– Notre atelier est basé sur une approche de programmation générative [55] et, de plus, utilise fortement des techniques standards (XML - *Extensible Markup Language*) afin de réduire les coûts de développement. Pour bénéficier des outils développés autour de ces standards, il devait être possible de les intégrer (*importer*) facilement en ne modifiant, ni le cœur du système, ni les outils générés. Ainsi notre plate-forme est ouverte et évolutive. En effet, quand ces standards et donc les outils associés évoluent, elle évolue aussi et de manière gratuite.
– Les outils (applications) produits ont vocation à être utilisés en dehors de notre atelier. Il fallait qu'ils puissent être facilement *exportés* vers d'autres applications. Il était donc important qu'ils soient indépendants de toute technologie de composants pour simplifier leur transposition vers les technologies de composants actuelles ou futures. Cette caractéristique donne une certaine latitude d'évolution à nos outils pour les prochaines années.

Toutes ces raisons nous ont donc conduits à opter pour une architecture par composants à modèle dédié à nos besoins, indépendant de toute technologie, et facilement transposable vers les technologies connues. En effet, la dernière raison énumérée ci-dessus nous interdisait de choisir une technologie particulière. Cette démarche correspond à la nouvelle stratégie défendue par l'OMG (*Object Management Group*), MDA (*Model-Driven Architecture*) [77], qui est basée sur une notion de transformation de modèles. Un des résultats importants

de notre démarche est de montrer, sur un exemple particulier, l'intérêt de définir un modèle indépendant de l'implantation (PIM - *Platform Independant Model*) et ensuite de proposer différentes transformations vers des modèles spécifiques (PSM - *Platform Specific Model*).

L'avantage principal de concevoir son propre modèle de composants est d'être en parfaite adéquation avec les besoins. Nos langages de description de composants sont ainsi adaptés et offrent un moyen déclaratif pour décrire précisément les composants et comment les assembler pour former une application, dans notre contexte. L'implantation effective du modèle (par génération de conteneurs à partir de ces descriptifs) prend en compte toutes les particularités de notre plate-forme, sans pour autant nuire à la lisibilité de ces descriptifs. Cette génération de conteneurs automatise la programmation des parties non-fonctionnelles des composants. Avoir un générateur facilite la prise en compte de nouveaux besoins ou de nouvelles technologies par simples modifications de ce dernier qui seront automatiquement reportées sur l'ensemble des applications. Cela facilite l'évolution de la plate-forme dans son ensemble.

Cette séparation entre le modèle et sa mise en œuvre rend possible l'utilisation de nos composants dans d'autres technologies. En effet, à partir de ces descriptifs, un outil de transformation génère les descriptions (ou interfaces) équivalentes pour les *Web Services*, les composants CORBA (*Common Object Request Broker Architecture*) ou encore les EJB (*Entreprise Java Bean*). Cette exportation nous a permis d'identifier les particularités, avantages et inconvénients de chaque modèle. Cela a confirmé que l'utilisation d'une de ces technologies aurait été préjudiciable.

En effet, certaines caractéristiques dynamiques de nos composants auraient été difficilement exprimables sans une spécialisation des composants en fonction des langages traités. Pour assurer une forte généricité à nos composants, il était important de pouvoir les adapter aux spécificités de chaque langage traité. Cela nous a imposé de prévoir l'ajout de nouveaux ports à un composant lors de son cycle de vie. Cette caractéristique était pertinente pour rendre nos composants de visualisation génériques.

L'article se décompose en quatre sections. La première section positionne nos travaux par rapport à des travaux similaires en indiquant quel est l'apport de notre démarche. Puis, les deuxième et troisième sections exposent, respectivement, le modèle de composants et sa mise en œuvre choisis en fonction des besoins de notre atelier logiciel. Enfin, la dernière section indique, en retour d'expérience, les caractéristiques que devraient posséder un modèle de composants et sa mise en œuvre afin d'obtenir une application facilement adaptable et évolutive.

4.2 Positionnement des travaux

Dans cette section, nous allons expliciter les contributions de notre approche par rapport aux nombreux travaux de recherche sur les composants et sur l'approche MDA. Depuis quelques années, l'objectif principal de ces travaux de recherche est d'assurer une meilleure ouverture et adaptabilité des applications, en proposant, par exemple, des modèles de composants plus ouverts [160] et adaptables dynamiquement. L'intérêt principal est d'obtenir une meilleure évolution des applications vis-à-vis des récents bouleversements

dans la conception du logiciel (applications distribuées).

Dans notre cadre, deux contraintes particulières émergent pour assurer un fort potentiel d'évolution à l'outil :

- avoir un modèle de composants indépendant de toute technologie pour pouvoir utiliser (exporter) les applications générées dans n'importe quel environnement quelle que soit sa technologie ;
- avoir des composants adaptables pour assurer la généricité de l'outil.

En étudiant les nombreux travaux de recherche sur le domaine des composants [117, 35, 13, 36, 137], nous nous sommes rendus compte que notre démarche semblait être particulière, car peu de ces travaux s'intéressent aux mêmes contraintes. Ces travaux peuvent être résumés dans les trois grandes orientations suivantes, complémentaires à nos préoccupations :

- l'ajout de nouveaux services (sécurité, transaction, persistence, etc.) sur un port donné ;
- la modification du cycle de vie d'un composant (session, entité, etc.). Par exemple, le remplacement d'un composant en cours d'exécution ;
- la modification des interactions entre composants (synchrone, événements, flux, etc.).

Pour positionner nos travaux, il nous semble donc important d'insister sur cette originalité, plus que sur les aspects communs à tout modèle de composants. De plus, sur ces aspects, la mise en œuvre de notre modèle reste encore très embryonnaire, car cela ne constitue pas, pour nous, une priorité. Notre démarche n'a pas la prétention de proposer un modèle générique (en remplacement des autres modèles) mais plutôt de montrer les avantages de définir un modèle de composants dirigé par les besoins.

En ce qui concerne la première contrainte, les ateliers logiciels ou les générateurs d'environnement de programmation [24, 97, 102, 110, 80, 72] s'en préoccupent peu. L'exportation des applications produites n'est pas facilitée. Ce point qui induit une forte dépendance des applications produites avec l'outil lui-même est souvent l'une des importantes critiques de ces outils. Plus généralement, ce soucis d'être indépendant vis-à-vis d'une technologie de composants a motivé l'approche MDA [77, 37] et la création d'un modèle UML (*Unified Modeling Language*) pour les composants. Mais tout cela est encore en gestation [59] et demandera certainement des approfondissements [26].

En ce qui concerne la deuxième contrainte, les travaux sur les composants adaptables, que nous avons étudiés, ne s'intéressent peu, en général, à étendre l'interface des composants (les conteneurs) par de nouveaux ports d'entrée ou de sortie comme nous le faisons. Dans [123], les auteurs ont aussi identifié ce besoin mais dans un cadre différent. Cette possibilité est souvent impossible [117], à cause de l'utilisation d'un typage fort pour la connexion ou d'une approche statique. Les travaux de recherche sur les composants s'intéressent plus à rendre adaptables les parties non-fonctionnelles des composants qu'à introduire de nouvelles fonctionnalités.

Notre modèle et sa mise en œuvre semblent être complets puisque ils semblent être en adéquation avec les rôles usuellement admis dans le processus de production d'applications à base de composants [116]. Par contre, l'originalité de notre approche apparaît clairement lorsque l'on s'intéresse aux aspects dynamiques des ports proposés et à l'indépendance vis-à-vis d'une technologie. Finalement, notre modèle caractérise les services spécifiques à notre fabrique d'applications orientées langages. Par comparaison, les mo-

dèles de composants, en particulier les industriels (EJB et CORBA), correspondent à des fabriques d'applications distribuées.

4.3 SMARTTOOLS et son modèle de composant abstrait

Cette section présente brièvement notre atelier logiciel et introduit son modèle de composants. Lors de la conception de ce modèle, nous nous sommes imposés deux objectifs : avoir une nette séparation entres les parties fonctionnelles et non-fonctionnelles, et avoir un modèle qui, bien que spécifique à nos besoins, soit «projetable» vers d'autres technologies existantes.

4.3.1 Brève présentation de SMARTTOOLS

Notre atelier logiciel est un générateur d'outils pour les langages de programmation ou métiers qui s'appuie fortement sur les technologies objets et XML [51, 132]. A partir de la description de structure de données d'un langage, il produit automatiquement (voir figure 4.1) des outils de manipulation des programmes (ou documents) pour ce langage. En interne, ces programmes sont représentés sous forme d'arbre de syntaxe abstraite. Avec les outils génériques offerts par l'atelier (tels que l'interface graphique, des vues graphiques génériques ou un analyseur syntaxique XML) et les outils générés pour un ou plusieurs langages, l'utilisateur peut rapidement obtenir un squelette d'application (environnement). Il peut ensuite l'enrichir, à l'aide de SMARTTOOLS, avec d'autres éléments spécifiques à son application comme :

- des vues graphiques sur les documents (*pretty-printers*) ;
- des analyseurs lexicaux et syntaxiques (*parsers*) ;
- des traitements particuliers (transformation, vérification et interprétation) liés à ce langage, écrits en Java avec le patron de conception visiteur (*visitor design pattern*) et la programmation par aspects, ou tout simplement des transformations écrites en XSL (*Extensible Stylesheet Language*) du W3C (*World Wide Web Consortium*).

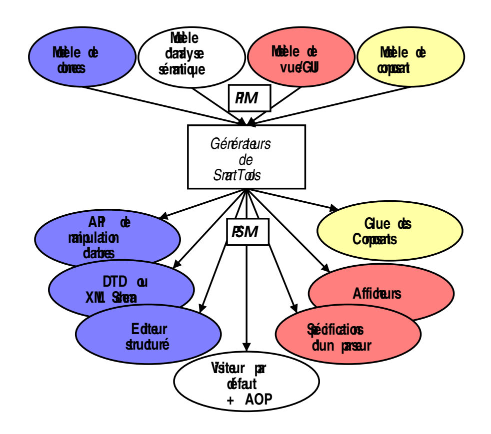


FIG. 4.1: Vue fonctionnelle de SMARTTOOLS

Notre atelier accepte, en entrée, les formalismes de description de structure de données du W3C (DTD - *Document Type Definition* - et XML schema), pour être ouvert et surtout pour avoir un vaste champ d'applications dans des domaines variés. Avec cette caractéristique, il n'est pas seulement limité à la conception d'outils pour les langages de programmation, mais pour tout type d'application où les structures de données sont définies à l'aide de ces formalismes.

Fonctionnement de base

Comme son fonctionnement a influencé le modèle de composants, il est très rapidement décrit pour mieux comprendre nos choix. L'atelier peut être utilisé en ligne de commande ou de manière interactive grâce à son interface graphique paramétrable. Dans les deux cas, la première opération effectuée sur le document à traiter est une analyse syntaxique pour obtenir l'arbre de syntaxe abstraite associé qui est la pierre angulaire de tous les outils produits. Puis, sur cette structure, peuvent être appliqués des outils d'analyses sémantiques (compilateurs), de transformation ou de visualisation.

Avoir une interface graphique implique un logiciel plus complexe puisqu'il faut gérer les interconnexions entre les documents, les outils, et les vues. De plus, dans notre cas, l'interface de l'atelier est hautement configurable afin de pouvoir l'adapter aux diverses applications produites. Elle est composée de fenêtres, d'onglets, de vues graphiques d'un ou plusieurs documents, et de menus contextuels (voir figure 4.2).

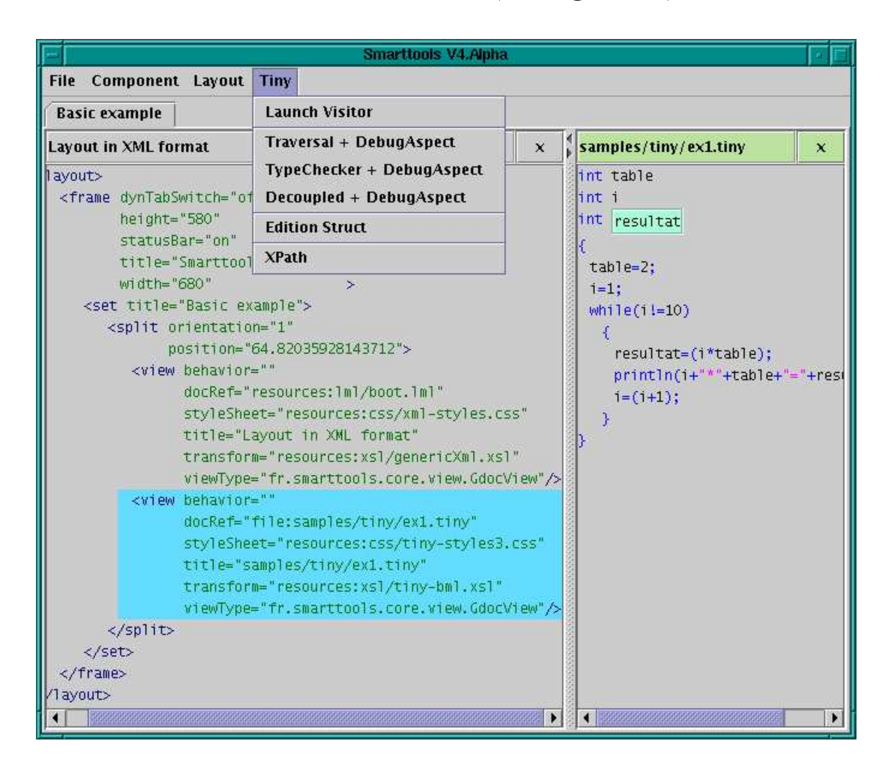


FIG. 4.2: Exemple d'interface graphique de SMARTTOOLS

Toute vue graphique dépend d'un document et est obtenue en appliquant une transformation à celui-ci. Le document et ses vues associées doivent rester consistants : toute modification incrémentale du document (changement de la position courante, ajout d'un nouveau nœud, etc) doit être répercutée sur ses vues selon le concept modèle-vue-contrôleur. Le fonctionnement des vues est générique quel que soit le langage d'appartenance des documents correspondants. Seules les données d'initialisation - le document traité, la transformation à appliquer, et la feuille de styles - diffèrent ainsi que les actions à ajouter au menu, spécifiques à chaque langage.

4.3.2 Modèle de composants

Les deux principaux types de composants manipulés dans notre plate-forme sont celui des *vues* et celui des *documents*. Lors de la conception du modèle de composants abstrait de notre application, nous devions prendre en compte les spécificités de chaque type. Pour cela, nous avons défini un langage de description de composant dont la syntaxe abstraite est décrite dans la figure 4.3.

```
component(formalism?, containerclass?, facadeclass?, parser*, lml?, behavior*,
          dependance*, attribute*, (input|output|inout)*)
   attributs : name, type?, extends?

formalism()
   attributs : name, file, dtd
containerclass()
   attributs : name
facadeclass()
   attributs : name, userclassname?
parser(extension)
   attributs : type, classname, extension, generator?, file?
lml()
   attributs : name, file
behavior()
   attributs : file
dependance()
   attributs : name, jar
attribute()
   attributs : name, javatype
input(parameters*)
   attributs : name, method
parameter()
   attributs : name, javatype
arg()
   attributs : type, ref
output(parameter*)
   attributs : name, method
inout(parameter*)
   attributs : name, method, outputname, outputmethod
```

FIG. 4.3: Syntaxe abstraite de notre modèle de composants

Des informations communes à ces deux types (*vues* et *documents*) peuvent être identifiées. Pour permettre la factorisation des descriptions, une notion simple d'héritage (attribut *extends*) a été introduite dans notre langage. Par exemple, le type de composant `abs-`

tractContainer de la figure 4.4 (voir figure 4.5 pour sa représentation graphique équivalente) est importé par tous nos types de composant.

Notre modèle de composant est composé des éléments de base suivants :

– le nom du composant (attribut name de l'élément component) ;
– le nom du type étendu (héritage - attribut extends de l'élément component) ;
– le nom du conteneur (containerclass) et celui de la façade (facadeclass) ;
– les dépendances vis-à-vis d'autres modules de code (dependance) ;
– les attributs de configuration de l'état du composant (attribute) ;
– les services (ports) en entrée ou en sortie avec leur mode de communication. Pour chaque service, on indique son nom logique, la méthode à appeler dans la partie métier (dans la façade du composant), et les paramètres de la méthode. Les modes de communication sont soit asynchrone (input ou output), soit synchrone (inout).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component name="abstractContainer">
  <input method="quit" name="quit"/>
  <output name="connectTo" method="connectTo">
    <parameter name="ref_src" javatype="java.lang.String"/>
    <parameter name="type_dest" javatype="java.lang.String"/>
    <parameter name="id_dest" javatype="java.lang.String"/>
    <parameter name="actions" javatype="java.util.HashMap"/>
  </output>
  <output name="exit" method="exit"/>
  <output name="initData" method="initData">
    <parameter name="inits" javatype="java.util.HashMap"/>
  </output>
  <inout name="requestData" method="request"
         output="initData" outputParameter="inits"/>
</component>
```

FIG. 4.4: Description du composant abstrait abstractContainer

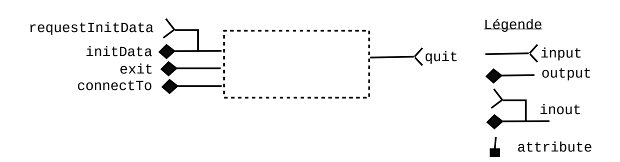


FIG. 4.5: Schéma du composant abstrait abstractContainer

La figure 4.6 donne un exemple de description de composant (celle du composant visualisation des graphes) et la figure 4.7 sa représentation graphique.

Composant vue

Pour éviter de définir un composant vue pour chaque langage et pouvoir proposer des vues génériques, un type de composant vue a été défini, indépendant de tout langage. Ce type décrit le comportement minimal et requis pour tous les composants vues. Sa représenta-

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component name="graph" type="graph" extends="abstractContainer">
  <containerclass name="GraphContainer"/>
  <facadeclass name="GraphFacade"/>
  <dependance name="koala-graphics" jar="koala-graphics.jar"/>
  <attribute name="nodeType" javatype="java.lang.String"/>
  <input name="addComponent" method="addNode">
    <parameter name="nodeName" javatype="java.lang.String"/>
    <parameter name="nodeColor" javatype="java.lang.String"/>
  </input>
  <input name="addEdge" method="addEdge">
    <parameter name="srcNodeName" javatype="java.lang.String"/>
    <parameter name="destNodeName" javatype="java.lang.String"/>
  </input>
</component>
```

FIG. 4.6: Description du composant Graphe

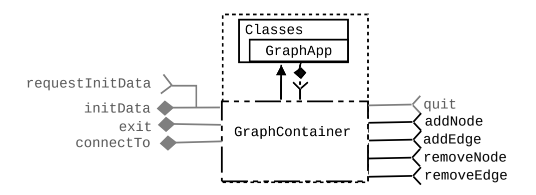


FIG. 4.7: Schéma du composant Graphe

tion graphique[2] est donnée en figure 4.8. Pour ajouter les services spécifiques à un langage, un mécanisme d'extension dynamique des composants vues a été introduit par l'intermédiaire de l'attribut de configuration *behavior* et du service `addBehavior` ; ce mécanisme sera décrit dans la section de mise en œuvre.

Composant document

Nos composants documents ont besoin, pour être construits et pour fonctionner, d'un ensemble d'informations supplémentaires telles que :

- le formalisme de base du langage (`formalism`) ;
- les diverses manières de lire un document (`parser`) ;
- les différentes vues associées par défaut (`lml`) ;
- les informations pour construire les menus spécifiques au langage dans l'interface utilisateur (`behavior`).

Comme les composants vues, les composants documents doivent respecter et réaliser un ensemble de services communs pour fonctionner. Ainsi, ils héritent tous d'un type de composant document standard (abstrait) décrit dans la figure 4.9. Par exemple, le service `launchVisitor`, commun à tous nos composants documents, permet de lancer un traitement (à base de visiteur) sur le document.

[2]Par la suite, seule la représentation graphique des types des composants est utilisée, plus lisible que la forme XML.

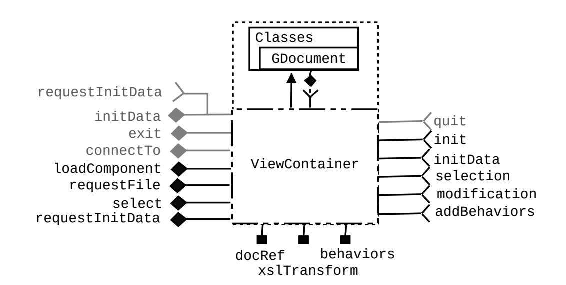


FIG. 4.8: Schéma du composant Vue

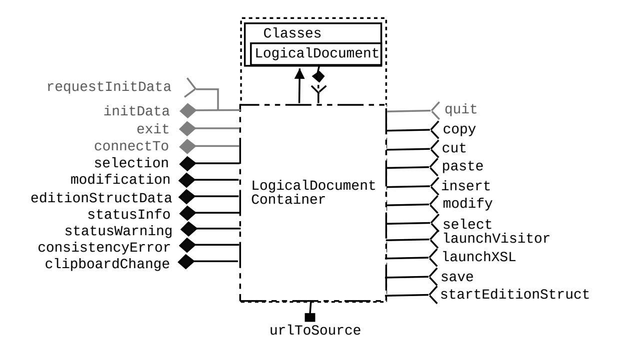


FIG. 4.9: Schéma du composant Document

Ces informations supplémentaires (comme le formalisme) sont aussi utilisées pour générer les parties du code métier du composant. À partir du formalisme, les structures de données du composant sont produites, utiles pour la construction d'un document (arbre). Le formalisme sert à valider les données complexes sérialisées en XML, lors des échanges entre le composant document et ses vues. Par exemple, la figure 4.10 montre le descriptif de composant du langage TINY (notre langage jouet).

Les informations associées à l'analyseur syntaxique précisent les différents protocoles de lecture du document en fonction des extensions des fichiers. Pour produire ces divers protocoles (élément `parser` de la figure 4.10), elles indiquent aussi le type de générateur à utiliser (attribut `generator`) et le fichier de spécifications correspondant (attribut `file`).

Pour qu'un composant document puisse fonctionner en mode interactif, il est nécessaire de préciser les composants vues à instancier qui vont permettre d'interagir avec lui à travers l'interface utilisateur. Cette information (`lml`) définit, de manière locale, pour chaque composant document, la topologie associée. La topologie de l'application résulte de la juxtaposition des topologies associées à chaque composant document instancié. Dans la section suivante, ce mécanisme (format `lml`) sera précisé.

Un composant document doit pouvoir aussi préciser quels sont les services spécifiques à ajouter aux composants vues (`behavior`) pour les adapter à ce dernier. Tous ces services, lorsqu'une de ses vues est sélectionnée, doivent être accessibles à travers les menus associés à la vue. Ce mécanisme doit pouvoir s'appliquer sur n'importe quel type de vue et, en particulier, sur les vues génériques fournies par défaut.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component name="tiny" group="document" extends="logicaldocument">
   <formalism name="tiny" file="tiny.absynt" dtd="tiny.dtd"/>
   <containerclass name="TinyContainer"/>
   <facadeclass name="TinyFacade"/>
   <parser type="xml" classname="tiny.parsers.TinyXMLParser"
           extention name="tinyxml"/>
   <parser type="plain-text" generator="ANTLR" file="tiny.g"
           classname="tiny.parsers.TinyParser" extention name="tiny"/>
   <lml  name="DEFAULT" file="resources:tiny-default.lml"/>
   <behavior file="resources:tiny-behaviors.xml"/>
</component>
```

FIG. 4.10: Description du composant du langage TINY

Finalement, ce modèle de composant a permis de rendre accessible à travers des spécifications déclaratives, l'ensemble des services et possibilités de notre plate-forme.

4.4 Mise en œuvre

Pour l'implantation du modèle, un générateur de conteneur et un gestionnaire de composants ont été conçus en fonction des contraintes de notre plate-forme. Pour chaque application construite avec SMARTTOOLS, un fichier de lancement précise les composants à charger et à instancier au démarrage comme, par exemple, l'interface graphique. Enfin, les mécanismes d'extension des composants et d'exportation sont présentés afin de rendre les composants, respectivement, indépendants de tout langage et facilement exportables dans une autre technologie de composant.

Le générateur

Le générateur produit le conteneur du composant à partir de son descriptif. Il peut éventuellement étendre la façade si cette dernière est incomplète (vérifiée par introspection). Cette extension sert à mettre en place le mécanisme de communication façade vers conteneur (proche du patron de conception *Observer*) pour les services en sortie (`output`). Ce générateur produit aussi une archive (un paquetage de déploiement) contenant l'ensemble des classes dont le conteneur et les descriptifs du composant. Cette phase de génération rend transparents les moyens de communication mis en œuvre dans le conteneur et automatise les tâches de création de paquetage pour chaque nouveau langage traité.

Le gestionnaire de composants

Le gestionnaire de composants charge les paquetages des composants et crée les instances en fonction d'une description de lancement de l'application (figure 4.14) et des demandes interactives des utilisateurs (par exemple, l'ouverture du premier document associé à un langage donné). Il mémorise l'ensemble des paquetages chargés et des instances créées. En particulier, il offre le service `connectTo` pour connecter deux composants

(figure 4.11). Ce service permet aussi la création des composants s'ils n'existent pas encore. Cela assure qu'un fichier ne corresponde qu'à un seul composant document. Pour les connexions, le gestionnaire utilise les descriptifs respectifs de ces composantn fonction des noms, les connecteurs de sortie (*vs.* d'entrée) sont mis en relation avec les connecteurs d'entrée (*vs.* de sortie) de l'autre composant, s'ils existent. Après ce processus de connexion, les deux instances de composant dialoguent directement entre elles sans passer par le gestionnaire. La figure 4.12 décrit l'ensemble des services proposés par ce gestionnaire.

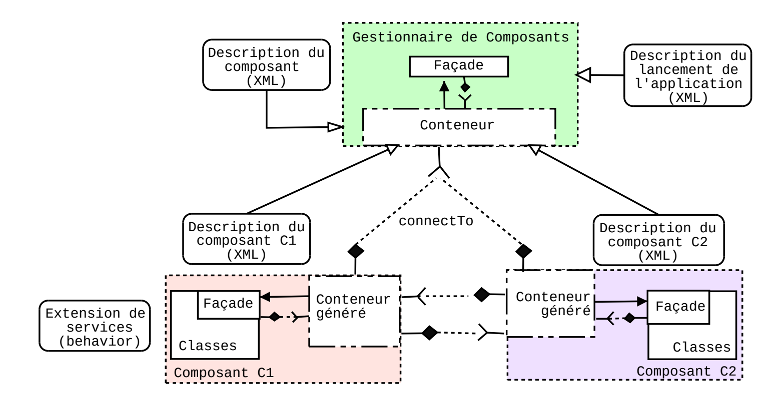


FIG. 4.11: Schéma de fonctionnement du gestionnaire

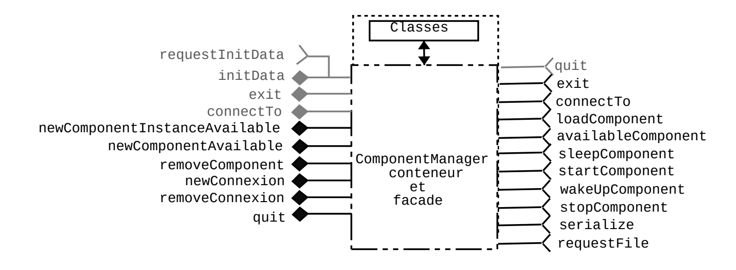


FIG. 4.12: Schéma du composant gestionnaire

Interface utilisateur

L'interface utilisateur est réalisée suivant le même schéma que pour la visualisation d'un document : c'est une vue sur un arbre qui décrit l'ensemble des vues et des documents ouverts à un instant donné. Les actions sur l'interface (ajout d'un nouvel onglet, fermeture d'une vue, etc.) correspondent à des actions d'édition sur cet arbre. L'état de l'interface peut ainsi être sauvegardé (*serialisable* en XML). Cette sauvegarde peut jouer le rôle de fichier de configuration de l'application (voir la figure 4.13). Pour chaque vue, on indique le chemin vers le document à visualiser, le type de vue, la transformation à effectuer sur le document pour obtenir les objets graphiques de cette vue et la feuille de style à appliquer sur ces objets. Pour un même document, il est donc possible d'avoir plusieurs vues graphiques associées.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE layout SYSTEM "file:resources/lml.dtd">
<layout>
  <frame title="Smarttools V4.Alpha" statusBar="on" width="680"
         height="580" dynTabSwitch="off">
    <set title="Basic example">
      <split orientation="1" position="68">
       <view title="Layout in XML format"
              behavior=""
              viewType="fr.smarttools.core.view.GdocView"
              docRef="resources:lml/boot.lml"
              styleSheet="resources:css/xml-styles.css"
              transform="resources:xsl/genericXml.xsl" />
       <view title="samples/tiny/ex1.tiny"
              behavior=""
              viewType="fr.smarttools.core.view.GdocView"
              docRef="file:samples/tiny/ex1.tiny"
              styleSheet="resources:css/tiny-styles3.css"
              transform="resources:xsl/tiny-bml.xsl" />
         </split>
    </set>
  </frame>
</layout>
```

FIG. 4.13: Exemple d'un arbre de l'interface graphique (`boot.lml`)

Lancement d'une application

Au démarrage de notre outil, le gestionnaire de composants est chargé et instancié par défaut. Puis, pour construire l'application, il va interpréter les actions d'un fichier de description de lancement. Les principales actions définies dans ce langage sont les suivantes :

– chargement d'un type de composant (`load_component`) ;
– création d'une instance (`start_component`) ;
– connexion entre deux composants (`connectTo`).

Par exemple, avec le fichier de lancement de la figure 4.14, il charge trois types de composant (`glayout`, `lml` et `tiny`) et établit une connexion entre lui-même et une instance du composant interface utilisateur (`glayout`). La création de cette dernière est paramétrée par les attributs de l'action `connectTo`. L'un de ces attributs (`docRef`) indique quel fichier de configuration de l'application est utilisé (*boot.lml* associé à l'application montrée dans la figure 4.2) pour instancier les composants documents et vues. L'ensemble des types de composant chargés (les rectangles) et les instances créées (les ovales) pour cet exemple sont décrits dans la figure 4.15.

Après le lancement de l'application (en fonction des fichiers des figures 4.13 et 4.14), les instances de composant créées sont, comme montrées dans la figure 4.15, les suivantes :

– une instance du gestionnaire de composants (une seule instance possible de ce type de composant) ;
– une instance de composant document `lml` pour l'interface utilisateur et une instance pour le langage `tiny` ;
– trois instances de composant vue (deux pour l'interface et une pour le programme `ex1.tiny`).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<application repository="file:stlib/"  library="file:lib/">
  <load_component jar="view.jar"  name="glayout"/>
  <load_component jar="lml.jar" name="lml"/>
  <load_component jar="tiny.jar" url="file:extralib/tiny.jar" name="tiny"/>
  <connectTo id_src="ComponentManager" type_dest="glayout">
    <attribute name="docRef" value="file:resources/lml/boot.lml"/>
    <attribute name="xslTransform" value="file:resources/xsl/lml2bml.xsl"/>
    <attribute name="behaviors" value="file:resources/behaviors/bootbehav.xml"/>
  </connectTo>
</application>
```

FIG. 4.14: Exemple de descriptif de lancement correspondant à l'application de la figure 4.2

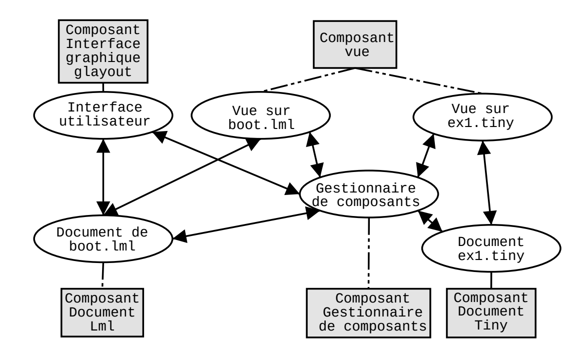


FIG. 4.15: Composants chargés et instances créées avec leurs connexions

Composants extensibles

Dans notre cadre, l'interface des composants de visualisation doit être extensible dynamiquement pour ajouter de nouveaux services. En effet, ceux-ci ne connaissent pas les services spécifiques des composants documents auxquels ils sont rattachés. Ils doivent donc être enrichis dynamiquement par ces services décrits dans un fichier d'extension du composant document (correspondant à l'élément `behavior` du descriptif du composant document). Ce fichier (voir figure 4.16) précise les éléments graphiques (menus et barre d'outils) à ajouter aux composants de visualisation (avec le service `addBehavior` ou l'attribut de configuration `behavior` des composants vues) et les nouvelles connexions à établir entre ces composants et le document.

Une autre caractéristique du modèle est la possibilité d'ajouter des services qui agissent à l'intérieur même de nos composants. En effet, notre plate-forme fournit une technique de programmation par aspect [33, 132] pour mieux spécifier (par *separation of concerns*) les traitements sur les arbres. Notre technique a l'avantage d'être totalement dynamique (sans transformation de programmes). L'effet de bord de cette possibilité est que les interfaces de nos composants documents doivent impérativement être extensibles. L'exemple type, que nous traitons déjà, est l'ajout d'un mode d'exécution pas-à-pas aux divers traitements, totalement réalisé à l'aide d'un aspect. Cet aspect utilise une vue graphique particulière et cela demande d'introduire dynamiquement de nouveaux services (dans les deux sens) entre

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<behaviors>
  <actions>
    <action name="Launch Visitor">
      <msg name="launchVisitor"
           chooser="fr.smarttools.core.document.LaunchVisitorDialogBox">
        <msgattr name="visitor"/>
        <msgattr name="aspect"/>
      </msg>
    </action>
    ...
  </actions>
  <menus>
    <menu name="File">
      <item action="Save Tiny"/>
    </menu>
    <menu name="Tiny">
      <item action="Launch Visitor"/>
      <item action="separator"/>
      <item action="Traversal + DebugAspect"/>
      ...
    </menu>
  </menus>
  ...
</behaviors>
```

FIG. 4.16: Exemple de services pour le langage TINY à rajouter à un composant vue (cf. menu de la figure 2)

le composant document et cette vue. Il nous reste encore à généraliser cette possibilité sur d'autres exemples plus complexes.

Détails de mise en œuvre

Comme notre outil est interactif, il était important que l'interface graphique ne soit pas bloquée lors de traitements sur les documents. Pour résoudre ce problème, chaque instance de composant est exécutée dans un processus indépendant (*Thread*) et le mode de communication est donc asynchrone (envoi d'événements stockés, à la réception, dans une file d'attente). Ces mécanismes sont transparents pour l'utilisateur car ils sont gérés au niveau du conteneur. Les mécanismes utilisés dans les conteneurs générés sont simples et efficaces (appel direct des méthodes de la façade, *multi-threading*, *listener*, etc.).

Les types des données échangées entre les composants ont volontairement été limités pour éviter que les composants ne soient obligés de connaître tous les types échangés. Pour être échangées, les données plus complexes doivent être sérialisées en XML. Cela oblige à définir la DTD correspondante pour les valider. Cette contrainte est implicitement réalisée dans notre cadre puisque nos données complexes, principalement des arbres, sont forcément associées à une DTD.

Pour faciliter l'implantation de nos générateurs en Java, il existe des références explicites à des types ou méthodes Java dans notre modèle. Cette dépendance au langage Java, au niveau du modèle, pourrait être levée en introduisant des tables de correspondance pour

divers langages de programmation.

Exportation des composants vers d'autres technologies

Pour valider notre approche, nous avons conçu un outil de transformation de nos descriptifs de composant vers les descriptions (ou interfaces) équivalentes pour les *Web Services*, les composants CORBA et les composants EJB. La figure 4.17 donne un aperçu des fichiers générés automatiquement par cet outil pour chaque technologie. Les détails techniques et particularités de cet outil sont décrits dans [163]. Notre composant de visualisation de graphe a ainsi été transformé automatiquement pour ces trois technologies. Cette transformation n'utilise que le descriptif du composant et aucun code source additionnel n'est à écrire. Par exemple pour les *Web Services*[3], cet outil traduit nos descriptifs de composant en des descriptifs *WSDL* (*Web Service Description Language*) équivalents. Par descriptif, il génère aussi le code source Java qui réalise les appels effectifs aux méthodes de la façade du composant (*SOAPBinding*). Ce code, normalement à la charge des développeurs, complète la génération de code source Java issu des outils associés aux *Web Services* comme AXIS. Ainsi, tous les services d'un composant peuvent être accessibles à travers un serveur Web comme *Tomcat*. Clairement, cet outil de transformation doit être considéré comme un prototype qui doit être approfondi sur des exemples plus complexes.

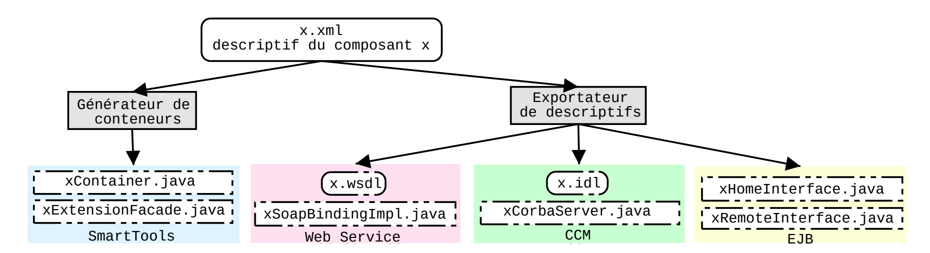


FIG. 4.17: Exemple de descriptif de composant

4.5 Évaluation de notre modèle

Comme il existe un nombre important de modèles et de concepts, il est assez difficile de faire une comparaison précise et exhaustive de notre approche. Pour cela, nous utiliserons le patron d'évaluation, issu des principaux modèles de composants, proposé par [116] qui présente un état de l'art des différents modèles et une taxonomie des rôles dans le processus de production d'applications à base de composants. Nous allons regarder l'adéquation de notre démarche avec cette taxonomie pour préciser les avantages ou inconvénients de notre modèle.

Analyse des besoins en terme d'applicatif et de composant Notre modèle est issu d'une analyse des besoins spécifiques à notre plate-forme et est donc adapté aux applications produites.

[3] L'une des technologies les plus simples de mise en œuvre et les plus proches de notre modèle.

Conception des types de composant Notre langage de description de composants permet de nous abstraire d'une technologie de composant particulière. Ce langage contient les trois aspects d'un composant : ses interfaces, ses propriétés, ainsi que la manière dont il coopère avec les autres composants.

Implantation des types de composant La partie non-fonctionnelle des composants est assurée par la génération automatique des conteneurs. Une partie des contraintes techniques (noms des conteneurs, des façades et des archives générés) est décrite dans les descriptifs des composants.

Diffusion des implantations de composants Notre générateur de conteneur prend en charge la constitution d'une archive (paquetage) pour chaque composant.

Assemblage des types de composant Les langages de déploiement et de description de l'interface graphique permettent de spécifier et de configurer l'assemblage des composants. Par contre, il n'est pas encore possible de considérer le résultat de cet assemblage comme un composant.

Déploiement des implantations de composant et des applicatifs Notre gestionnaire de composants joue le rôle de déploiement de composants. Il prend en charge la création de nouvelles instances et la recherche des instances existantes. Dans notre modèle, le déploiement n'est pas encore configurable en terme de structure d'accueil (par exemple, pour une version répartie sur différentes machines).

Utilisation des instances de composant et des applicatifs La découverte des services offerts par un composant s'effectue dans notre modèle exclusivement par l'intermédiaire des descriptifs des composants (format XML). Nous utilisons une introspection sur ce format neutre.

Au vue de notre expérience et de cette taxonomie des rôles, nous pouvons établir une liste de caractéristiques essentielles que tout modèle de composants devrait posséder, de notre point de vue, pour être flexible et évolutif. Ces caractéristiques sont les suivantes :

– Les descriptifs de composant doivent être indépendants de tout langage et de toute technologie, et proches des besoins de l'application. Cela induit une meilleure lisibilité des services offerts par les différents types de composant vu que le modèle est adapté aux besoins (modèle conçu en fonction des besoins établis pendant l'analyse). L'utilisation des EJB implique *de facto* une approche client/serveur, CORBA une dépendance vis-à-vis de son protocole de communication (ORB - *Object Request Broker*) et les *Web-Services* une spécialisation Internet des composants. La construction de l'architecture d'une application devient aisée (même pour des débutants) si les composants à assembler ont des descriptifs compréhensibles (avec le vocabulaire du métier). L'interconnexion des composants est facilitée et est sans introspection car le générateur utilise des descriptifs au format neutre. Cette indépendance permet aussi de traduire ces descriptifs vers d'autres modèles de composants (WSDL, IDL, etc) dans le but de faciliter l'exportation des composants dans une application à technologie de composant différente.
– Les conteneurs doivent être produits automatiquement à l'aide d'un générateur. Le développeur se focalise seulement sur la partie métier, la valeur ajoutée, du composant lors du développement de ce dernier. Cette séparation entre la partie métier et la

partie non-fonctionnelle est bénéfique car elle assure une indépendance de la partie métier vis-à-vis d'une technologie. Cette indépendance rend plus facile la projection vers d'autres technologies. Les dépendances d'un composant sont limitées aux entités (bibliothèques) utiles à son fonctionnement, sans faire référence à la technologie de composant utilisée. De plus, une application conçue à l'aide d'un générateur de conteneur a un meilleur potentiel d'adaptation et d'évolution. En effet, tout nouveau besoin ou toute nouvelle technologie (de communication, par exemple) peut, très rapidement, être intégré dans l'application, par simple modification de ce générateur. Par exemple, dans notre cadre, la prise en compte d'une version répartie devrait être facilement gérée par modification de notre générateur. Avoir son propre générateur permet aussi de produire du code spécifique à ses besoins (par exemple, pour effectuer des tests unitaires). La maintenance de la «glue» de l'application des composants est ainsi gérée automatiquement (*refactoring* d'applications) par le générateur.

– Il est important d'avoir une topologie locale, dynamique et configurable. Une architecture variable permet l'ajout de nouveaux composants, à la demande, en fonction des besoins. Dans notre plate-forme, il est possible de brancher dynamiquement un composant d'observation des messages échangés. Les composants doivent pouvoir localement décider de leurs connexions avec d'autres composants. Dans notre cadre, si un nouveau composant vue est créé, c'est lui qui va décider dynamiquement sa connexion avec le composant document associé. La construction de la topologie d'une application doit être décentralisée au niveau des composants eux-mêmes. Cette topologie doit aussi pouvoir être sauvegardée dans un fichier de déploiement pour donner la possibilité de relancer l'application dans le même état.
– Les composants doivent pouvoir communiquer directement entre eux. Cela évite les goulets d'étranglement au niveau d'un serveur d'application (gestionnaire de composants) et simplifie la mise en œuvre des communications (le gestionnaire de composants ne traite que des services liés à la création, au chargement, et à la connexion). Les composants restent autonomes.
– Pour la souplesse de l'application, il est important d'avoir des composants adaptables aussi bien pour les parties non-fonctionnelles que pour les parties fonctionnelles. Pour les parties non-fonctionnelles, de telles possibilités ont été clairement identifiées dans le cadre des intergiciels [117]. Pour les parties fonctionnelles, notre expérience montre clairement la nécessité d'un tel besoin.

4.6 Conclusion

La principale motivation de ce travail était de définir un modèle de composants en adéquation avec nos besoins et non d'en faire un modèle générique. Ce modèle est un moyen déclaratif de décrire l'architecture de SMARTTOOLS, indépendamment d'une technologie de composant et d'un langage de programmation. Les technologies existantes, comme les composants CORBA ou EJB, nous ont semblé être assez éloignées de nos préoccupations ou ne répondaient pas complètement à nos attentes. Pour autant, notre modèle s'appuie fortement sur les mêmes concepts de base mais sa mise en œuvre est adaptée à notre mode

de fonctionnement. Nous avons préféré définir notre propre modèle, indépendamment des technologies existantes, tout en permettant une transformation vers celles-ci.

L'une des principales caractéristiques de SMARTTOOLS est d'être basée sur une approche par génération de code à partir de spécifications, non seulement pour les parties non-fonctionnelles des composants (conteneurs) mais aussi pour les parties fonctionnelles (API de manipulation des arbres, visiteur de parcours par défaut, etc.). Finalement, notre modèle caractérise les services spécifiques à notre fabrique d'applications orientées langages. Par comparaison, les modèles de composants, en particulier les industriels (EJB et CORBA), correspondent plus à des fabriques d'applications distribuées.

La principale contribution de nos travaux est de démontrer que ce concept de fabrique d'applications peut, d'une part, s'appliquer à différents niveaux dans une application (même sur les parties métiers) et, d'autre part, qu'elle doit être accessible à travers des modèles indépendants de toute technologie. Nous sommes plus convaincus de l'intérêt d'une approche par famille d'applications (fabrique) qu'à une approche de modèle générique de composants. En effet, il nous semble préférable de définir un modèle adapté au vocabulaire du métier sous-jacent. De plus, pour répondre aux besoins d'ouvertures et d'évolutions des applications, il est important que la fabrique (les générateurs) et les applications générées soient elles-mêmes adaptables.

Par son auto-utilisation dans sa propre construction, SMARTTOOLS offre un cadre pour tester concrètement les solutions proposées. C'est avant tout une plate-forme d'expérimentation pour nos futurs travaux de recherche, en particulier sur la composition d'aspects et de services pour les composants. De plus, les extensions naturelles de notre modèle (composant hiérarchique ou exécution répartie) offrent des perspectives et des problèmes (édition collaborative de documents) très intéressants.

Quatrième partie

Le dossier

Chapitre 5

Le dossier

5.1 Valorisations et transferts technologiques

A Rocquencourt

La réalisation du système FNC-2 a été le produit phare de notre projet à Rocquencourt, le pilier justifiant nos nombreuses participations à des projets européens :

- 86-88 : le projet ESPRIT COCOS (Components for Future Computing Systems). Son but était la conception d'une architecture ouverte pour stations de travail multi-usages.
- 89-91 : le projet ESPRIT (2105) MULTIWORK (MULTImedia WORKstation) avec Antoine Rizk. L'objectif était la conception de stations de travail multimédia à un prix relativement bas.
- 91-95 : le projet ESPRIT (5399) COMPARE (COMpiler generation for PARallel machinEs) avec comme résultat la plate-forme générique de compilation COSY. Étant donné l'ampleur et la nature du projet, j'ai organisé le travail d'une équipe de 6 personnes pendant 4 ans. J'étais arrivé à intégrer le système FNC-2 à la plate-forme COSY qui est devenue une belle réussite industrielle (voir le site http://www.ace.nl/, comme framework pour les codes embarqués) ;
- 89-90 Attrisem (Action nationale) avec Isabelle Attali pour établir une passerelle entre FNC-2 et Centaur dans le but de réaliser une évaluation efficace de la Sémantique Naturelle.

A Sophia Antipolis

Dans le projet OASIS, l'outil SMARTTOOLS a été impliqué dans les contrats de recherche suivants :
- Action Dyade SMARTTOOLS avec un financement de Microsoft de 150 K$ pour deux ans
- Contrat Ministère Formavie (appel d'offre Oppidum) (1999 - 2001, montant de 1 360 KF HT) sur le thème "modélisations de machines virtuelles embarquées". Partenariat avec Bull/CP8 et Schlumberger.
- Contrat de recherche avec Bull/CP8 (2000), (montant de 150 KF HT) sur le thème "Editeur et vérificateur pour Java Card sous Windows".
- Contrat de recherche avec Bull/CP8 (1998 - 2001, montant de 1 362 KF HT) sur le thème "Environnement de développement et vérification pour Java Card".

En 2002, l'outil SMARTTOOLS a été impliqué dans les contrats suivants :
- 2002-2003, projet IST (Information Society Technologies), QUESTION-HOW piloté par Daniel Dardailler, W3C. SMARTTOOLS est utilisé comme un démonstrateur des technologies du W3C ;
- contrat de licence d'utilisation de 50 KF avec une PME, DYNAM-IT.

En 2002, deux projets RNTL ont été soumis dans le cadre de SMARTTOOLS :
- XLUC, projet exploratoire sur trois ans sur le thème "Transformations réversibles de conception UML (*Unified Modeling Language*) contractualisées et vérifiables". Partenariat avec Université de Vannes (Valoria), Université de Nice (I3S) et Softeam (refusé).
- IntelliSurf, projet pré-compétitif sur deux ans sur le thème " Plate-forme d'accélération des livraisons de contenu Web" ; partenariat avec DYNAM-IT (une PME) et un client GERDOSS. (accepté)

Cette année 2003, avec l'outil SMARTTOOLS, je suis impliqué dans :
- une soumission d'un projet RNTL de type plate-forme, ModaThèque, avec plusieurs partenaires (voir le résumé [17]) sur le thème MDA (*Model Driven Architecture*).
- L'action de Recherche et Développement [1], SYNTAX, sur le thème du traitement des documents électroniques.
- une COLOR [6] sur la Protection dans les langages de programmation, avec l'équipe OCL de l'université de Nice et l'équipe RPO du LIRMM de l'université de Montpellier.

5.2 Encadrement de thèses

J'ai encadré les 8 thèses (les 5 premières avec MARTIN JOURDAN) suivantes :

1. Bruno Marmol. *La parallélisation et l'optimisation mémoire dans l'évaluation des grammaires attribuées*. PhD thesis, Université d'Orléans, 1994.

ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/theses/marmol.ps.gz.
Ingénieur de Recherche à l'INRIA (Grenoble)

2. Gilles Roussel. *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées*. PhD thesis, Département d'Informatique, Université Paris 6, March 1994.
ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/theses/roussel.ps.gz.
MdC à l'université de Marne-la-Vallée
3. Carole Le Bellec. *La généricité et les grammaires attribuées*. PhD thesis, Département de Mathématiques et d'Informatique, Université d'Orléans, 1993.
ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/theses/lebellec.ps.gz.
4. Aziz Souah. *Contribution à la sémantique déclarative des systèmes de transformation d'arbres attribués*. PhD thesis, Université d'Orléans, November 1990.
Ingénieur chez MetaWare
5. Catherine Julié. *Optimisation de l'espace mémoire pour l'évaluation de grammaires attribuées*. PhD thesis, Université d'Orléans, September 1989.
MdC à l'université d'Orléans
6. Carine Courbis. *Contribution à la programmation générative. Application dans le générateur SmartTools : technologies XML, programmation par aspects et composants*. PhD thesis, Université de Nice, Decembre 2002.
ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/theses/Courbis2002.ps.gz.
Post-Doc University College London
7. Loic Correnson. *Sémantique Equationnelle*. PhD thesis, l'Ecole Polytechnique, April 2000.
ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/theses/Correnson2000.ps.gz.
Ingénieur chez Trusted-Logic
8. Etienne Duris. *Contribution aux relations entre les grammaires attribuées et la programmation fonctionnelle*. PhD thesis, Université d'Orléans, 1998.
ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/theses/Duris98.ps.gz.
MdC à l'université de Marne-la-Vallée

5.3 Encadrement de stages (DEA,DESS, maîtrise)

J'ai co-encadré avec Martin Jourdan les stages suivants :
(http://www-sop.inria.fr/oasis/personnel/Didier.Parigot/stages.html)

- En 1986, J Bonnet (DEA d'Orléans) et C. Julié (DEA d'Orléans) ;
- En 1987, C. Ayrault (DEA d'Orléans) et A. Souah (DEA d'Orléans) ;
- En 1988, O. Durin (stage d'option de l'École Polytechnique) et S. Taouil (DEA d'Orléans) ;
- En 1989, O. Durin (DEA), et E. Planes (DEA d'Orléans) et C. Le Bellec (DEA d'Orléans) ;
- En 1990, C. Zylberman (telecom), B. Marmol (DEA d'Orléans), D. Devillar (DEA d'Orléans) ;

- En 1991, J.P JOUVE (stage d'option de l'École Polytechnique) et R. GOMEZ (DEA) ;
- En 1992, P. BAZET (DEA), B. AMILIEN ;
- En 1993, P. ROUZIER et B. AMILIEN et H. BENVEL (DEA) ;
- En 1994, É. DURIS (DEA d'Orléans) ;
- En 1995, G. LE BÂTARD (DESS) ;
- En 1996, S. LEIBOVITSCH (DEA sémantique et preuves) et L. CORRENSON (stage d'option de l'École Polytechnique)
- En 1997, L. CORRENSON (DEA sémantique et preuves)

Dans le projet OASIS, j'ai encadré les étudiants suivants :
- En 2000, A. BERGEL, T. ABBONDANZA, J.L. BAUDOIN et D. NADÉ (projet et stage d'été)
- En 2001 T. ABBONDANZA (DESS) et J.G VARIAMPARAMBIL (Indian Institute of Technology, Kanpur) ;
- En 2002, O. CHABROL (DESS), P. FARRUGIA (DESS), S. ZIANE(DEA) et J.G. VARIAMPARAMBIL ;
- En 2003, J. BAUDRY (DESS) et D. ROTOVEI (DEA)

5.3.1 Encadrement d'ingénieurs

Dans le cadre des projets contractuels, j'ai encadré techniquement les ingénieurs experts, les étudiants et les chercheurs suivants :
- En 1989-1991, A. RIZK (IE) pour les projets COCOS et MULTIWORK
- En 1991-94, T. GAÁL (IE), M. MAZAUD (Ingénieur INRIA), A. DESPLAND (MdC), F. THOMASSET (DR inria), L. COGNARD (Doctorante), P. CANALDA (Doctorant) pour le projet Compare ;
- En 1999-01, A. FAU, P. DEGENNE, F. CHALAUX, J. FILLON, C. PASQUIER (**CR2 au Laboratoire de Physiologie des Membranes Cellulaires, NICE**) et C. HELD pour les projets Java Card, Formavie et SMARTTOOLS ;
- En 2002, A. FAU (**ingénieur à Ilog**) et P. DEGENNE (**ingénieur de recherche au CIRAD**) pour le projet européen QUESTION-HOW.

5.3.2 Liste des rapports de DEA ou DESS

Voici la liste des rapports de DEA ou DESS correspondants aux stages :

1. Bernard Amilien. Interfaces graphiques interactives sous XWindows pour le système de traitement de grammaires attribuées FNC-2. Rapport de stage, Institut d'Ingéniérie Informatique de Limoges, September 1992.
2. Bernard Amilien. Réalisation d'un complément interactif de trace de circularité dans le compilateur de grammaires attribuées FNC-2. Rapport de stage de fin d'études, Institut d'Ingéniérie Informatique de Limoges, 1993.
3. Christine Ayrault. Implantation en C des structures de données du langage OLGA. rapport de DEA, University d'Orléans, September 1987.

4. Philippe Bazet. Pattern-matching en priorité spécifique sur des types avec constructeurs commutatifs. Rapport de DEA, Université de Paris 11, Orsay, September 1992.
5. Hervé Benvel. Réalisation d'un front-end pascal avec le système FNC-2. Rapport de DEA, Université de Paris 6, September 1993.
6. Joël Bonnet. Étude des principaux langages de description de grammaires attribuées et spécification d'un nouveau langage basé sur des grammaires abstraites. rapport de DEA, University d'Orléans, September 1986.
7. Olivier Chabrol. Passerelle entre les méta-modèles (uml) et les formalismes de syntaxe abstraite à l'aide des formalismes du w3c (xmi). rapport de DESS, Marseille, June 2002.
8. Loïc Correnson. Généricité dans les grammaires attribuées. Rapport de stage d'option, École Polytechnique, 1996.
9. Loïc Correnson. Programmation polytypique avec les grammaires attribuées. Rapport de DEA, Université de Paris VII, September 1997.
10. David Devillard. Amélioration de l'implantation en C des évaluateurs d'attributs produits par FNC-2. Rapport de DEA, Université d'Orléans, September 1990.
11. Olivier Durin. Traduction en OLGA d'une grammaire attribuée écrite en lisp. rapport de stage d'option, École Polytechnique, Palaiseau, July 1988.
12. Olivier Durin. Génération en le_lisp d'évaluateurs d'attributs spécifiés en olga. rapport de magistère, École Normale Supérieure, Paris, September 1989.
13. Etienne Duris. Transformation de grammaires attribuées pour des mises à jour destructives. Rapport de DEA, Université d'Orléans, September 1994.
14. Patrice Farrugia. Etude d'un langage de transformation, cosynt. Rapport de DESS, Marseille, June 2002.
15. Rémi Forax. Le langage chocolat. Rapport de stage de maîtrise d'informatique, Université de Marne la Vallée, July 1997.
16. José Garcia, Martin Jourdan, and Antoine Rizk. An implementation of PARLOG using high-level tools. In Commission of the European Communities—Directorate General XIII, editor, *ESPRIT '87 : Achievements and Impact*, pages 1265–1275. North-Holland, Amsterdam, November 1987. Brussels.
17. Roberto Gomez Cardenas. Amélioration de l'implantation en C des évaluateurs d'attributs produits par FNC-2. Rapport de DEA, Université de Paris 7, September 1991.
18. Martin Jourdan, Bruno Marmol, and Didier Parigot. Experiments with a Real Parallel Attribute Evaluator. En préparation pour soumission, 1994.
19. Jean-Philippe Jouve. Réalisation du décompilateur d'arbres attribués du système FNC-2 : PPAT. Rapport de stage d'option, École Polytechnique, 1991.
20. Catherine Julié. Optimisation de l'espace mémoire pour les compilateurs générés selon la méthode d'évaluation OAG : étude des travaux de kastens et propositions d'améliorations. rapport de DEA, Dépt. d'Informatique, University d'Orléans, September 1986.

21. Carole Le Bellec. Spécification de règles sémantiques manquantes. rapport de DEA, Dépt. d'Informatique, University d'Orléans, September 1989.

22. Gilles Le Bâtard. Réalisation dans le système FNC-2 d'un traducteur vers ML. rapport de stage de maîtrise, IFI, Université de Marne-la-Vallée, July 1995.

23. Stéphane Leibovitsch. Relations entre la sémantique dénotationnelle et les grammaires attribuées. Rapport de DEA, Université de Paris VII, September 1996.

24. Bruno Marmol. Évaluateurs d'attributs parallèles sur multi-processeurs à mémoire partagée. rapport de DEA, University d'Orléans, September 1990.

25. Étienne Planes. PPAT : un décompilateur d'arbres attribués pour le système FNC-2. rapport de DEA, Dépt. d'Informatique, University d'Orléans, September 1989.

26. Christophe Roudet. Visualisation graphique incrémentale par évaluation d'attributs. Stage de DEA informatique de l'ESSI, Université NICE, 1994.

27. Gilles Roussel. Élimination de suites de règles de copies dans les grammaires attribuées. 1992.

28. Gilles Roussel. Méta-composition des grammaires attribuées. rapport de magistère, École Normale Supérieure, Paris, September 1992.

29. Philippe Rouzier. Réalisation d'une interface entre les systèmes Centaur, FNC-2 et syntax. Rapport de stage de DESS "Systèmes et communication homme-machine", Université de Paris 11, September 1993.

30. Aziz Souah. Système de transformation d'arbres attribués : étude des principaux systèmes et spécification d'un nouveau système. rapport de DEA, University d'Orléans, September 1987.

31. Souad Taouil. Étude et implantation des grammaires couplées par attributs dans le système FNC-2. rapport de DEA, Dépt. d'Informatique, University d'Orléans, September 1988.

32. Joseph George Variamparambil. Getting smarttools and visualstudio.net to talk to each other using soap and web services. Technical report, INRIA, 2001.

33. Joseph George Variamparambil. Enabling smarttools components with component technologies :webservices, corba and ejbs. Technical report, INRIA, 2002.

34. Bruno Vivien. Etude et réalisation d'un compilateur E-LOTOS à l'aide du générateur de compilateurs SYNTAX/FNC-2. Diplome d'ingenieur CNAM en informatique, Conservatoire National des Arts et Métiers, Décembre 1997.

35. Saïda Ziane. Passerelle entre les formalismes (schema) du w3c et la programmation par objets. rapport de DEA, Versailles, June 2002.

36. Catherine Zylberman. Réalisation du constructeur d'arbre abstrait (ATC) du système de grammaire attribuée FNC-2 au dessus de l'analyseur lexico-syntaque lex-yacc. Rapport de stage de troisième année, Telecom Paris, 1990.

5.4 Publications

1. Didier Parigot. Mise en œuvre des grammaires attribuées : transformation, évaluation incrémentale, optimisations. thèse de 3ème cycle, University de Paris-Sud, Orsay, September 1987.
2. Didier Parigot. *Transformations, Évaluation Incrémentale et Optimisations des Grammaires Attribués : Le Système FNC-2*. PhD thesis, Université de Paris-Sud, Orsay, 1988.

Stage de DEA

3. Didier Parigot. Un système interactif de trace des circularités dans une grammaire attribuée et optimisation du test de circularité. rapport de DEA, University de Paris-Sud, Orsay, September 1985.

Prix

4. All. Apport des technologies Objets et XML aux applications des cartes à puce et du commerce électronique. 2001. label de la Recherche, association Telecom Valley.

Journaux

5. Carine Courbis, Pascal Degenne, Alexandre Fau, and Didier Parigot. Un modèle abstrait de composants adaptables.. accepté à la revue Objet, parution en 2004, ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Parigot04a.pdf.
6. Carine Courbis, Pascal Degenne, Alexandre Fau, and Didier Parigot. L'apport des technologies XML et Objets pour un générateur d'environnements : SmartTools. accepté à la revue Objet, parution en 2003, ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Parigot02a.pdf.

Editeurs

7. Didier Parigot and Marjan Mernik, editors. *Second Workshop on Attribute Grammars and their Applications WAGA'99*, Amsterdam, The Netherlands, March 1999. ETAPS'99, INRIA rocquencourt. Satellite event of ETAPS'99.
8. Didier Parigot and Marjan Mernik, editors. *Third Workshop on Attribute Grammars and their Applications WAGA'00*, Ponte de Lima, Portugal, July 2000. MPC'2000, INRIA rocquencourt. Satellite event of MPC'2000.
9. Didier Parigot, Marjan Mernik, and Mark van den Brand, editors. *Workshop on Language Descriptions, Tools and Applications*, Genova, Italy, April 2001. ETAPS'2001, Electronic Notes in Theoretical Computer Science (ENTCS). Satellite event of ETAPS'2001.
10. Marjan Mernick and Didier Parigot, editors. *Attribute Grammars and Their Applications*, volume 24. Informatica, Slovenia, 2000.

Conférences Internationales

11. Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, and Claude Pasquier. SmartTools : a Generator of Interactive Environments Tools. In Reinhard Wilhelm, editor, *International Conference on Compiler Construction CC'01*, volume 2027 of *Lect. Notes in Comp. Sci.*, Genova, Italy, April 2001. ETAPS'2001, Electronic Notes in Theoretical Computer Science (ENTCS). Tools Demonstrations at CC'01.

12. Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Declarative program transformation :
a deforestation case-study. In Gopalan Nadathur, editor, *Principles and Practice of Declarative Programming PPDP'99*, volume 1702 of *Lect. Notes in Comp. Sci.*, pages 353–369, Paris, France, October 1999.

13. Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Equational semantics. In Agostino Cortesi and Gilberto Filé, editors, *Symposium Static Analysis SAS'99*, volume 1694 of *Lect. Notes in Comp. Sci.*, pages 264–283, Venice, Italy, 1999.

14. Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Attribute grammars and functional programming deforestation. In *Fourth International Static Analysis Symposium – Poster Session*, Paris, France, September 1997.

15. Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic Attribute Grammars. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of *Lect. Notes in Comp. Sci.*, pages 122–136, Aachen, September 1996. Springer-Verlag.

16. Gilles Roussel, Didier Parigot, and Martin Jourdan. Coupling Evaluators for Attribute Coupled Grammars. In Peter A. Fritzson, editor, *5th Int. Conf. on Compiler Construction (CC' 94)*, volume 786 of *Lect. Notes in Comp. Sci.*, pages 52–67, Edinburgh, April 1994. Springer-Verlag.

17. Carole Le Bellec, Martin Jourdan, Didier Parigot, and Gilles Roussel. Specification and Implementation of Grammar Coupling Using Attribute Grammars. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '93)*, volume 714 of *Lect. Notes in Comp. Sci.*, pages 123–136, Tallinn, August 1993. Springer-Verlag.

18. Martin Jourdan and Didier Parigot. Internals and Externals of the FNC-2 Attribute Grammar System. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lect. Notes in Comp. Sci.*, pages 485–504. Springer-Verlag, Prague, 1991.

19. Martin Jourdan and Didier Parigot. Application Development with the FNC-2 Attribute Grammar System. In Dieter Hammer and Michael Albinus, editors, *Compiler Compilers '90*, volume 477 of *Lect. Notes in Comp. Sci.*, pages 11–25. Springer-Verlag, Schwerin, 1990.

20. Martin Jourdan and Didier Parigot. Techniques for Improving Grammar Flow Analysis. In Neil Jones, editor, *European Symp. on Programming (ESOP '90)*, volume 432

of *Lect. Notes in Comp. Sci.*, pages 240–255, Copenhague, 1990. Springer-Verlag.

21. Catherine Julié and Didier Parigot. Space Optimization in the FNC-2 Attribute Grammar System. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lect. Notes in Comp. Sci.*, pages 29–45, Paris, September 1990. Springer-Verlag.
22. Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conf. on Programming Languages Design and Implementation*, pages 209–222, White Plains, NY, June 1990. Published as *ACM SIGPLAN Notices*, 25(6).
23. Martin Jourdan, Carole Le Bellec, and Didier Parigot. The Olga Attribute Grammar Description Language : Design, Implementation and Evaluation. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lect. Notes in Comp. Sci.*, pages 222–237, Paris, 1990. Springer-Verlag.

Workshop internationaux

24. Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Joel Fillon, Christophe Held, Didier Parigot, and Claude Pasquier. Aspect and XML-oriented Semantic Framework Generator SmartTools. In *Second Workshop on Language Descriptions, Tools and Applications, LDTA'02*. ETAPS'2002, Electronic Notes in Theoretical Computer Science (ENTCS), 2002. Satellite event of ETAPS'2002.
25. Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Joël Fillon, Didier Parigot, Claude Pasquier, and Claudio Sacerdoti Coen. SmartTools : a development environment generator based on XML technologies. In *XML Technologies and Software Engineering*, Toronto, Canada, 2001. ICSE'2001, ICSE workshop proceedings.
26. Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Joel Fillon, Christophe Held, Didier Parigot, and Claude Pasquier. Workshop on language descriptions, tools and applications. In *Aspect and XML-oriented Semantic Framework Generator SmartTools*. ETAPS'2002, Electronic Notes in Theoretical Computer Science (ENTCS), 2002. Satellite event of ETAPS'2002.
27. Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Generic programming by program composition (position paper). In *Workshop on Generic Programming*, Marstrand, Sweden, June 1998. conjunction with MPC'98.
28. Martin Jourdan and Didier Parigot. The FNC-2 system : Advances in attribute grammar technology. In O. M. Tammepuu, editor, *Procs. of the Soviet-French Symposium Informatika '89*, pages 94–118, Tallinn, May 1989. See also : rapport RR-834, INRIA, Rocquencourt (April 1988).

Workshop ou conférence français

29. Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, and Joseph Variamparambil. Un modèle de composants pour l'atelier de développement SmartTools. In *Systèmes à composants adaptables et extensibles*, Octobre 2002.

30. Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Composition symbolique. In *journées du GDR de programmation*, Rennes, November 1997.

31. Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Composition symbolique. In *Journées Francophones des Langages Applicatifs*, Come, Italie, February 1998.

32. Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Schéma générique de développement par composition. In *Approches Formelles dans l'Assistance au Développement de Logiciel AFADL'98*, Poitiers - futuroscope, 1998.

33. Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Grammaires attribuées et folds : Opérateurs de contrôle Génériques. In *journées du GDR de programmation*, Orléans, 1996.

34. Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Grammaires attribuées et folds : opérateurs de contrôle génériques. In *Journées Francophones des Langages Applicatifs*, Dolomieu, France, January 1997.

35. Didier Parigot, Etienne Duris, Gilles Roussel, and Martin Jourdan. Les grammaires attribuées : un langage fonctionnel déclaratif. In *journées du GDR de programmation*, Grenoble, November 1995.

36. Didier Parigot, Etienne Duris, Gilles Roussel, and Martin Jourdan. Les grammaires attribuées : un langage fonctionnel déclaratif. In *Journées Francophones des Langages Applicatifs*, pages 263–279, Val-Morin, Québec, January 1996.

Rapports de recherche

37. Isabelle Attali and Didier Parigot. Integrating Natural Semantics and Attribute Grammars : the Minotaur System. Rapport de recherche 2339, INRIA, 1994.

38. Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. A generic framework for genericity. English version of, 1998.

39. Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Symbolic composition. Technical Report 3348, INRIA, January 1998.

40. Loic Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. How to deforest in accumulative parameters ? Technical Report 3608, INRIA, January 1999.

41. Etienne Duris, Didier Parigot, and Martin Jourdan. Mises à jour destructives dans les grammaires attribuées. Rapport de recherche 2686, INRIA, October 1995.

42. Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Attribute grammars and folds : Generic control operators. Rapport de recherche 2957, INRIA, August 1996.

43. Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Structure-directed genericity in functional programming and attribute grammars. Rapport de Recherche 3105, INRIA, February 1997.

44. Martin Jourdan and Didier Parigot. More on speeding up circularity tests for attribute grammars. rapport RR-828, INRIA, Rocquencourt, April 1988.

45. Catherine Julié and Didier Parigot. Space Optimization in the FNC-2 Attribute Grammar System. Rapport de recherche 1165, INRIA, 1990.

46. Didier Parigot, Etienne Duris, Gilles Roussel, and Martin Jourdan. Attribute grammars : a declarative functional language. Rapport de Recherche 2662, INRIA, October 1995.

47. Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic Attribute Grammars. Rapport de recherche 2881, INRIA, May 1996.

48. Gilles Roussel, Didier Parigot, and Martin Jourdan. Static and Dynamic Coupling Attribute Evaluators. Rapport de recherche 2670, INRIA, October 1995.

Divers

49. Didier Parigot and Martin Jourdan. A complete bibliography on attribute grammars. `http ://www-rocq.inria.fr/oscar/www/fnc2/AGabstract.html` Updated regularly. Contains around 1000 references to papers on Attribute Grammars. INRIA, France.

50. Martin Jourdan and Didier Parigot. *The FNC-2 System User's Guide and Reference Manual*. INRIA, Rocquencourt, 1.9 edition.

Bibliographie

[1] Action de recherche & développement : Syntax. http://www-sop.inria.fr/oasis/personnel/Didier.Parigot/SYNTAX/.

[2] ANTLR - ANother Tool for Language Recognition. http://www.antlr.org/.

[3] Apache ant - the jakarka project (apache). http://jakarta.apache.org/ant.

[4] AspectJ - Aspect-Oriented Programming (AOP) for Java. http://aspectj.org.

[5] Batik SVG toolkit (apache). http://xml.apache.org/batik/.

[6] Color : Protection dans les langages de programmation. http://www-sop.inria.fr/oasis/personnel/Didier.Parigot/COLOR/.

[7] CORBA 3.0 CCM. http://www.omg.org/.

[8] CUP - LALR Parser Generator for Java. http://www.cs.princeton.edu/$~$appel/modern/java/CUP/.

[9] Eclipse. http://www.eclipse.org/.

[10] Entreprise java beans. http://java.sun.com/products/ejb.

[11] http ://www.w3.org/. The World Wide Web Consortium.

[12] Java Compiler Compiler (JavaCC) - The Java Parser Generator (Sun). http://www.webgain.com/products/java_cc.

[13] Jonas. http ://jonas.objectweb.org/.

[14] Model management group. http://research.microsoft.com/db/ModelMgt/.

[15] MOF 2.0 query / views / transformations rfp. http://www.omg.org/techprocess/meetings/schedule/MOF_2.0_Query_View%_Transf._RFP.html.

[16] Omg. http://www.omg.org/.

[17] projet RNTL modathèque. http://www-sop.inria.fr/oasis/personnel/Didier.Parigot/MDA/.

[18] UML - unified modeling language. http://www.uml.org.

[19] XMI - XML Metadata interchange (OMG). http://www.omg.org/technology/documents/formal/xmi.htm.

[20] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1) :68–88, 1997.

[21] Henk Alblas and Bořivoj Melichar, editors. *Attribute Grammars, Applications and Systems*, volume 545 of *Lect. Notes in Comp. Sci.* Springer-Verlag, New York–Heidelberg–Berlin, June 1991. Prague.

[22] Isabelle Attali. Compiling typol with attribute grammars. In Pierre Deransart, Bernard Lorho, and Jan Maluszynski, editors, *Programming Languages Implementation and Logic Programming*, volume 348 of *Lecture Notes in Computer Science*, pages 252–272. Springer-Verlag, New York–Heidelberg–Berlin, May 1988. Orléans.

[23] Isabelle Attali and Didier Parigot. Integrating Natural Semantics and Attribute Grammars : the Minotaur System. Rapport de recherche 2339, INRIA, 1994.

[24] Don Batory, Bernie Lofaso, and Smaragdakis. JTS : A tool suite for building genvoca generators. In *5th International Conference in Software Reuse*, June 1998.

[25] F. Bellegarde. Program Transformation and Rewriting. In *Proceedings of the fourth conference on Rewriting Techniques and Applications*, volume 488 of *Lect. Notes in Comp. Sci.*, pages 226–239. Springer-Verlag, 1991.

[26] Nicolas Belloir, Jean-Michel Bruel, and Franck Barbier. Formalisation de la relation tout-partie : application à l'assemblage de composants logiciels. In *Actes des Journées composants : Flexibilité du système au langage (JC'2001)*, Besançon, France, October 2001.

[27] J.A. Bergstra and P. Klint. The discrete time ToolBus – A software coordination architecture. *Science of Computer Programming*, 31(2-3) :205–229, July 1998.

[28] J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3) :205–229, July 1998.

[29] P. Bernstein. Applying model management to classical meta data problems, 2003.

[30] Patrick Borras. Ppml reference manual and compiler implementation. In *ESPRIT Project 348 GIPE—Third Annual Review Report*. |INRIA|, January 1988.

[31] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. CENTAUR : the system. *SIGSOFT Software Eng. Notes*, 13(5) :14–24, November 1988.

[32] Noury M. N. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. In *Technique et Sciences Informatiques*, volume 20, page 505 à 528. Hermès, 2001. http://www.emn.fr/dept_info/perso/ledoux/Publis/tsi01.pdf.

[33] Noury M. N. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. In *Technique et Sciences Informatiques*, volume 20, page 505 à 528. Hermès, 2001.

[34] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. MSL : a model for W3C XML Schema. *Computer Networks (Amsterdam, Netherlands : 1999)*, 39(5) :507–521, August 2002.

[35] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. Recursive and dynamic software composition with sharing. In *In Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga (Spain), June 2002.

[36] Eric Bruneton and Michel Riveill. Experiments with JavaPod, a platform designed for the adaptation of non-functional properties. In *Metalevel Architectures and Sepa-*

ration of Crosscutting Concerns, REFLECTION 2001, volume 2192 of *LNCS*, pages 52–72, Kyoto, Japan, sept 2001.

[37] Jean Bézivin. From Object composition to Model Transformation with MDA. In *TOOLS USA*. IEEE TOOLS-39, 2001.

[38] Jean Bézivin and Xavier Blanc. MDA : Vers un important changement de paradigme en génie logiciel. *Développeur référence v2.16*, 2002.

[39] Bruno Chabrier, Vincent Lextrait, and Paul Franchi-Zannettacci. GIGAS : a graphical interface generator from attribute specifications. In *Actes des Journées Int. "Le Génie Logiciel et ses Applications"*, pages 1265–1283. EC2, Paris, December 1988. Toulouse.

[40] P. Clements. A Survey of Architecture Description Languages. In *Proc. 8th ACM/IEEE Int. Workshop on Software Specification and Design (IWSSD)*, pages 16–25, Germany, 1996. IEEE Computer Society Press.

[41] The COMPARE Consortium. Repport of toolselection taskforce. draft, Compare Consortium, august 1991. confidential.

[42] The COMPARE Consortium. fsdl. draft, Compare Consortium, novembre 1994. confidential.

[43] The COMPARE Consortium. Using pagode and fnc-2 in compare. draft, Compare Consortium, july 1994. confidential.

[44] The COMPARE Consortium. Cosy compilers overview of their construction and operation
with glossary, list of acronyms and index. draft, Compare Consortium, march 1995. confidential.

[45] Loïc Correnson. Généricité dans les grammaires attribuées. Rapport de stage d'option, École Polytechnique, 1996. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Rapport/correnso.ps.gz.

[46] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Schéma générique de développement par composition. In *Approches Formelles dans l'Assistance au Développement de Logiciel AFADL'98*, Poitiers - futuroscope, 1998. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Correnson98b.ps.gz.

[47] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Declarative program transformation :
a deforestation case-study. In Gopalan Nadathur, editor, *Principles and Practice of Declarative Programming PPDP'99*, volume 1702 of *Lect. Notes in Comp. Sci.*, pages 353–369, Paris, France, October 1999. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Correnson99b.ps.gz.

[48] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Equational semantics. In Agostino Cortesi and Gilberto Filé, editors, *Symposium Static Analysis SAS'99*, volume 1694 of *Lect. Notes in Comp. Sci.*, pages 264–283, Venice, Italy, 1999.

[49] Loic Correnson. Equational Semantics. *Informatica (Solvenia)*, 1999. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Correnson99c.ps.gz.

[50] Loic Correnson. *Sémantique Equationnelle*. PhD thesis, l'Ecole Polytechnique, April 2000.

[51] Carine Courbis, Pascal Degenne, Alexandre Fau, and Didier Parigot. L'apport des technologies XML et Objets pour un générateur d'environnements : Smarttools. *revue L'Objet, numéro spécial XML et les Objets*, 2002. à paraître, ftp ://ftp-sop.inria.fr/oasis/publications/2002/smartobjet02.pdf.

[52] Carine Courbis, Pascal Degenne, Alexandre Fau, and Didier Parigot. Un modèle de composants pour l'atelier de développement SMARTTOOLS. In *Journée systèmes à composants adaptables et extensibles*, Grenoble, France, October 2002. ftp://ftp-sop.inria.fr/oasis/personnel/Carine.Courbis/smartcompo02.pdf.

[53] Carine Courbis, Pascal Degenne, Alexandre Fau, and Didier Parigot. L'apport des technologies XML et Objets pour un générateur d'environnements : SmartTools. accepté à la revue Objet, parution en 2003, ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Parigot02a.pdf, 2003.

[54] Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, and Joseph Variamparambil. Un modèle de composants pour l'atelier de développement SmartTools. In *Systèmes à composants adaptables et extensibles*, Octobre 2002. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Parigot02g.pdf.

[55] Carine Courbis and Didier Parigot. Different ways of using Generative Programming to develop an application, 2003.

[56] S. Cranefield. UML and the Semantic Web. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*, 2001. http://www.semanticweb.org/SWWS/program/full/paper1.pdf.

[57] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming : Methods, Techniques, and Applications*. Addison-Wesley, June 2000. ISBN 0201309777 chapter Aspect-Oriented Decomposition and Composition http://www-ia.tu-ilmenau.de/~czarn/aop/.

[58] E. Dashofy, A. van der Hoek, and R. Taylor. An infrastructure for the rapid development of xml-based architecture description languages, 2002.

[59] Miguel de Miguel, Jean Jourdan, and Serge Salicki. Practical experiences in the application of MDA. *Lecture Notes in Computer Science*, 2460 :128– ? ?, 2002.

[60] Pierre Deransart and Martin Jourdan, editors. *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*. Springer-Verlag, New York–Heidelberg–Berlin, September 1990. Paris.

[61] Annie Despland, Monique Mazaud, and Raymond Rakotozafy. Using rewriting techniques to produce code generators and proving them correct. Rapport RR-1046, |INRIA|, June 1989.

[62] Etienne Duris. Transformation de grammaires attribuées pour des mises à jour destructives. Rapport de DEA, Université d'Orléans, September 1994. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Rapport/etienne.ps.gz.

[63] Etienne Duris, Didier Parigot, and Martin Jourdan. Mises à jour destructives dans les grammaires attribuées. Rapport de recherche 2686, INRIA, October 1995.

[64] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Attribute grammars and folds : Generic control operators. Rapport de recherche 2957, INRIA, August 1996.

[65] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Grammaires attribuées et folds : Opérateurs de contrôle Génériques. In *journées du GDR de programmation*, Orléans, 1996.

[66] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Structure-directed genericity in functional programming and attribute grammars. Rapport de Recherche 3105, INRIA, February 1997.

[67] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable attribute grammars : Support for modularity in translator design and implementation. In *19th ACM Symp. on Principles of Programming Languages*, pages 223–234, Albuquerque, NM, January 1992. ACM press.

[68] Gilberto Filé. Classical and incremental attribute evaluation by means of recursive procedures. In Paul Franchi-Zannettacci, editor, *11th Coll. on Trees in Algebra and Programming (CAAP '86)*, volume 214 of *Lecture Notes in Computer Science*, pages 112–126. Springer-Verlag, New York–Heidelberg–Berlin, March 1986. Nice.

[69] M. M. Fokkinga, J. Jeuring, L. Meertens, and E. Meijer. A translation from attribute grammars to catamorphisms. *The Squiggolist*, 2(1) :20–26, 1991.

[70] Rémi Forax, Etienne Duris, and Gilles Roussel. Java multi-method framework. In *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00)*, November 2000.

[71] Pascal Fradet and Mario Südholt. AOP : towards a generic framework using program transformation and analysis. In *International Workshop on Aspect-Oriented Programming at ECOOP, 1998*, 1998.

[72] E. Gagnon and L. Hendren. SableCC - An Object-Oriented Compiler Framework. In *In Proceedings of TOOLS 1998 - 26th International Conference and Exhibition*, pages 140–154, August 1998.

[73] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[74] Harald Ganzinger. Some principles for the development of compiler from denotational language definitions. Bericht TUM-INFO-8006, Institut für Informatik, Tech. University München, May 1980.

[75] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 157–170, Montréal, June 1984. ACM press. Published as *ACM SIGPLAN Notices*, 19(6).

[76] Robert Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25 :355–423, 1988.

[77] OMG Staff Strategy Group and Richard Soley. Model-Driven Architecture. Technical report, OMG, November 2000. ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf.

[78] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In Andreas Paepcke, editor, *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428. ACM Press, October 1993.

[79] Görel Hedin and Eva Magnusson. JastAdd—a Java-based system for implementing front ends. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science, LDTA'01 First Workshop on Language Descriptions, Tools and Application, ETAPS'2001*, volume 44, Genova, Italy, April 2001. Elsevier Science Publishers. http://www.cs.lth.se/home/Gorel_Hedin/LDTA/RefJava.Submitted.pdf.

[80] Görel Hedin and Eva Magnusson. JastAdd—a Java-based system for implementing front ends. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[81] H. Hosoya and B. C. Pierce. "XDuce : A Typed XML Processing Language". In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000.

[82] IBM. Bean Markup Language. http://www.alphaworks.ibm.com/formula/bml.

[83] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lect. Notes in Comp. Sci.*, pages 68–114. Springer-Verlag, 1996.

[84] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Func. Prog. Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag, New York–Heidelberg–Berlin, September 1987. Portland.

[85] Martin Jourdan. *Des bienfaits de l'analyse statique sur la mise en œuvre des grammaires attribuées*. Mémoire d'habilitation, Département de Mathématiques et d'Informatique, Université d'Orléans, 1992. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/theses/habilit-martin.ps.gz.

[86] Martin Jourdan. Presentation of dynamic attribute grammars, 1996. http ://www-rocq.inria.fr/oscar/FNC-2/Presentation/DynAG.slides.ps.gz.

[87] Martin Jourdan, Bruno Marmol, and Didier Parigot. Experiments with a Real Parallel Attribute Evaluator. En préparation pour soumission, 1994.

[88] Martin Jourdan and Didier Parigot. *The FNC-2 System User's Guide and Reference Manual*. INRIA, Rocquencourt, 1.9 edition. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/f2manual.ps.gz.

[89] Martin Jourdan and Didier Parigot. Techniques for Improving Grammar Flow Analysis. In Neil Jones, editor, *European Symp. on Programming (ESOP '90)*, volume 432 of *Lect. Notes in Comp. Sci.*, pages 240–255, Copenhague, 1990. Springer-Verlag. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/esop90-t.ps.gz.

[90] Martin Jourdan and Didier Parigot. Internals and Externals of the FNC-2 Attribute Grammar System. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Evaluation Methods*, volume 545 of *Lect. Notes in Comp. Sci.*, pages 485–504, New York–Heidelberg–Berlin, June 1991. Springer-Verlag. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/fnc2-t.ps.gz.

[91] Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conf. on Programming Languages Design and Implementation*, pages 209–222, White Plains, NY, June 1990. Published as *ACM SIGPLAN Notices*, 25(6) ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/sgpln90-t.ps.gz.

[92] Jean-Philippe Jouve. Réalisation du décompilateur d'arbres attribués du système FNC-2 : PPAT. Rapport de stage d'option, École Polytechnique, 1991. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Rapport/ppat.ps.gz.

[93] Catherine Julié. Optimisation de l'espace mémoire pour les compilateurs générés selon la méthode d'évaluation OAG : étude des travaux de kastens et propositions d'améliorations. rapport de DEA, Dépt. d'Informatique, University d'Orléans, September 1986.

[94] Catherine Julié. *Optimisation de l'espace mémoire pour l'évaluation de grammaires attribuées*. PhD thesis, Université d'Orléans, September 1989.

[95] Catherine Julié and Didier Parigot. Space Optimization in the FNC-2 Attribute Grammar System. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lect. Notes in Comp. Sci.*, pages 29–45, Paris, September 1990. Springer-Verlag. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/optim-t.ps.gz.

[96] Catherine Julié and Didier Parigot. Space Optimization in the FNC-2 Attribute Grammar System. Rapport de recherche 1165, INRIA, 1990.

[97] Uwe Kastens, Peter Pfahler, and Matthias Jung. The eli system. In Kai Koskimies, editor, *Compiler Construction CC'98*, volume 1383 of *Lect. Notes in Comp. Sci.*, portugal, April 1998. Springer-Verlag. tool demonstration.

[98] Gregor Kiczales. Aspect-oriented programming : A position paper from the xerox PARC aspect-oriented programming project. In Max Muehlhauser, editor, *Special Issues in Object-Oriented Programming*. 1996.

[99] Gregor Kiczales and Jim des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[100] Gregor Kiczales, Jim Hugunin, Mik Kersten, John Lamping, Cristina Lopes, and William G. Griswold. Semantics-Based Crosscutting in AspectJ. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, 2000.

[101] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming*

11th European Conference, Jyväskylä, Finland, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[102] Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2) :176–201, 1993.

[103] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2) :127–145, June 1968. Correction : *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[104] Tobias Kuipers and Joost Visser. Object-Oriented Tree Traversal with JJForester. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[105] Carole Le Bellec. Spécification de règles sémantiques manquantes. rapport de DEA, Dépt. d'Informatique, University d'Orléans, September 1989.

[106] Carole Le Bellec. *La généricité et les grammaires attribuées*. PhD thesis, Département de Mathématiques et d'Informatique, Université d'Orléans, 1993. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/theses/lebellec.ps.gz.

[107] Carole Le Bellec, Martin Jourdan, Didier Parigot, and Gilles Roussel. Specification and Implementation of Grammar Coupling Using Attribute Grammars. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '93)*, volume 714 of *Lect. Notes in Comp. Sci.*, pages 123–136, Tallinn, August 1993. Springer-Verlag. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/couple-AG-plilp93.ps.gz.

[108] Gilles Le Bâtard. Réalisation dans le système FNC-2 d'un traducteur vers ML. rapport de stage de maîtrise, IFI, Université de Marne-la-Vallée, July 1995.

[109] Stéphane Leibovitsch. Relations entre la sémantique dénotationnelle et les grammaires attribuées. Rapport de DEA, Université de Paris VII, September 1996. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Rapport/leibovit.ps.gz.

[110] Karl J. Lieberherr and Doug Orleans. Preventive program maintenance in Demeter/Java. In *Proceedings of the 19th International Conference on Software Engineering*, pages 604–605. ACM Press, May 1997.

[111] Peter Lipps, Ulrich Möncke, Matthias Olk, and Reinhard Wilhelm. Attribute (re)evaluation in OPTRAN. *Acta Informatica*, 26 :213–239, 1988. See also : ESPRIT PROSPECTRA Project Report S.1.3 - R.4.0, University des Saarlandes, Saarbrücken (1986).

[112] J. Malenfant and P. Cointe. Aspect-Oriented Programming versus Reflection. Technical report, Ecole des Mines de Nantes, 1996.

[113] Jacques Malenfant, Christophe Dony, and Pierre Cointe. A Semantics of Introspection in a Reflective Prototype-Based Language. *Lisp and Symbolic Computation*, 9(2/3) :153–180, May/June 1996.

[114] Bruno Marmol. Évaluateurs d'attributs parallèles sur multi-processeurs à mémoire partagée. rapport de DEA, University d'Orléans, September 1990.

[115] Bruno Marmol. *La parallélisation et l'optimisation mémoire dans l'évaluation des grammaires attribuées*. PhD thesis, Université d'Orléans, 1994. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/theses/marmol.ps.gz.

[116] R. Marvie and M.-C. Pellegrini. Modèles de composants, un état de l'art. *Numéro spécial de L'Objet*, 8(3), 2002.

[117] Raphäel Marvie, Philippe Merle, Jean-Marc Geib, and Mathieu Vadet. OpenCCM : une plate-forme ouverte pour composants CORBA. In *Actes de la seconde Conférence Française sur les Systèmes d'Exploitation (CFSE'2)*, Paris, France, Avril 2001.

[118] Illustrations With Mda. Two approaches in system modeling and their.

[119] Microsoft. The .NET platform, 2001. http://www.microsoft.com/net/.

[120] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 279–303, Lisbonne, Portugal, June 1999. Springer-Verlag.

[121] Frank Neven and Jan Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. In *PODS '98. Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 11–17. ACM press, 1998.

[122] Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice-Hall, 1995.

[123] Audrey Occello, Mireille Blay-Fornarino, Anne-Marie Dery, and Michel Riveill. Vers une adaptation dynamique cohérente des composants. In *Actes des Journées : Systèmes à composants adaptables et extensibles*, Grenoble, France, October 2002.

[124] Jukka Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2) :196–255, June 1995.

[125] J. Palsberg and K. Tao. Java Tree Builder, 1997. http://www.cs.purdue.edu/jtb.

[126] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, Vienna, Austria, August 1998.

[127] Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. In Hanne Riis Nielson, editor, *European Symposium on Programming*, pages 280–295, Linkoping, Sweden, 1996. Springer Verlag.

[128] Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2) :264–292, March 1995.

[129] Didier Parigot. Un système interactif de trace des circularités dans une grammaire attribuée et optimisation du test de circularité. rapport de DEA, University de Paris-Sud, Orsay, September 1985.

[130] Didier Parigot. Mise en œuvre des grammaires attribuées : transformation, évaluation incrémentale, optimisations. thèse de 3ème cycle, University de Paris-Sud, Orsay, September 1987.

[131] Didier Parigot. *Transformations, Évaluation Incrémentale et Optimisations des Grammaires Attribués : Le Système FNC-2*. PhD thesis, Université de Paris-Sud, Orsay, 1988.

[132] Didier Parigot, Carine Courbis, Pascal Degenne, Alexandre Fau, Claude Pasquier, Joël Fillon, Christophe Help, and Isabelle Attali. Aspect and XML-oriented Semantic Framework Generator : SmartTools. In *ETAPS'2002, LDTA workshop*, Grenoble, France, April 2002. Electronic Notes in Theoretical Computer Science (ENTCS). ftp://ftp-sop.inria.fr/oasis/publications/2002/smartldta02.pdf.

[133] Didier Parigot, Etienne Duris, Gilles Roussel, and Martin Jourdan. Les grammaires attribuées : un langage fonctionnel déclaratif. In *Journées Francophones des Langages Applicatifs*, pages 263–279, Val-Morin, Québec, January 1996. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/jfla96.ps.gz.

[134] Didier Parigot and Martin Jourdan. A complete bibliography on attribute grammars. http://www-rocq.inria.fr/oscar/www/fnc2/AGabstract.html Updated regularly. Contains around 1000 references to papers on Attribute Grammars. INRIA, France.

[135] Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic Attribute Grammars. Rapport de recherche 2881, INRIA, May 1996.

[136] Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic Attribute Grammars. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of *Lect. Notes in Comp. Sci.*, pages 122–136, Aachen, September 1996. Springer-Verlag. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/plilp96.ps.gz.

[137] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC : A flexible solution for aspect-oriented programming in Java. *Lecture Notes in Computer Science*, 2192 :1– ? ?, 2001.

[138] Étienne Planes. PPAT : un décompilateur d'arbres attribués pour le système FNC-2. rapport de DEA, Dépt. d'Informatique, University d'Orléans, September 1989.

[139] Steven P. Reiss. Simplifying Data Integration : The Design of the Desert Software Development Environment. In *Proceedings of the 18th International Conference on Software Engineering*, pages 398–407. IEEE Computer Society Press, 1996.

[140] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, pages 42–48. ACM press, Pittsburgh, PA, April 1984. Joint issue with Software Eng. Notes 9, 3.Published as ACM SIGPLAN Notices, volume 19, number 5.

[141] Christophe Roudet. Visualisation graphique incrémentale par évaluation d'attributs. Stage de DEA informatique de l'ESSI, Université NICE, 1994.

[142] Gilles Roussel. Élimination de suites de règles de copies dans les grammaires attribuées. 1992.

[143] Gilles Roussel. Méta-composition des grammaires attribuées. rapport de magistère, École Normale Supérieure, Paris, September 1992.

[144] Gilles Roussel. *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées*. PhD thesis, Département d'Informatique, Université de Paris 6, March 1994. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/theses/roussel.ps.gz.

[145] Gilles Roussel, Didier Parigot, and Martin Jourdan. Coupling Evaluators for Attribute Coupled Grammars. In Peter A. Fritzson, editor, *5th Int. Conf. on Compiler Construction (CC' 94)*, volume 786 of *Lect. Notes in Comp. Sci.*, pages 52–67, Edinburgh, April 1994. Springer-Verlag. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/couplingevaluatorAG.ps.gz.

[146] Gilles Roussel, Didier Parigot, and Martin Jourdan. Static and Dynamic Coupling Attribute Evaluators. Rapport de recherche 2670, INRIA, October 1995.

[147] K. E. Schauser, D. E. Culler, and S. C. Goldstein. Separation constraint partitioning - A new algorithm for partitioning non-strict programs into sequential threads. In *Proc. Principles of Programming Languages (POPL'95)*, San Francisco, CA, January 1995.

[148] Klaus E. Schauser and Seth C. Goldstein. How much non-strictness do lenient programs require ? In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 216–225, La Jolla, CA, USA, June 1995. ACM Press.

[149] Jérôme Siméon and Philip Wadler. The essence of XML. *ACM SIGPLAN Notices*, 38(1) :1–13, January 2003.

[150] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In D. Sannella, editor, *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 485–500. Springer-Verlag, 1994.

[151] Aziz Souah. Système de transformation d'arbres attribués : étude des principaux systèmes et spécification d'un nouveau système. rapport de DEA, University d'Orléans, September 1987.

[152] Aziz Souah. *Contribution à la sémantique déclarative des systèmes de transformation d'arbres attribués*. PhD thesis, Université d'Orléans, November 1990.

[153] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[154] F. Tartanoglu, V. Issarny, N. Levy, and A. Romanovsky. Dependability in the web service architecture, 2002.

[155] Croap team. *A Centaur Tutorial*. INRIA Sophia-Antipolis, July 1994. http://www-sop.inria.fr/croap/centaur/tutorial/main/main.html.

[156] OMG Architecture Board MDA Drafting Team. Model Driven Architecture - A Technical Perspective. Technical report, OMG, July 2001. http://www.omg.org/cgi-bin/doc?ormsc/01-07-01.pdf.

[157] Delphine Terrasse. *Vers un environnement de développement de preuves en Sémantique Naturelle*. PhD thesis, Ecole Nationale des Ponts et Chaussées (ENPC), October 1995.

[158] G. Tremblay and G. R. Gao. The impact of laziness on parallelism and the limits of strictness analysis. In *High Performance Functional Computing*, pages 119–133, April 1995.

[159] Guy Tremblay. *Parallel implementation of lazy functional languages using abstract demand propagation*. Phd thesis, McGill University, Montreal Canada, November 1994.

[160] Mathieu Vadet and Philippe Merle. Les containers ouverts dans les plate-forme à composant. In *Actes des Journées composants : Flexibilité du système au langage (JC'2001)*, Besançon, France, October 2001.

[161] A. van Deursen and P. Klint. Little languages : Little maintenance ? *Journal of Software Maintenance*, 1998.

[162] Joseph George Variamparambil. Getting smartTools and visualstudio.NET to talk to each other using SOAP and webServices. Technical report, INRIA, 2001.

[163] Joseph George Variamparambil. Enabling smartTools components with component technologies :webServices, CORBA and EJBs. Technical report, INRIA, 2002.

[164] Joseph George Variamparambil. Enabling SMARTTOOLS components with component technologies : Web-Services, CORBA and EJBs. Technical report, INRIA, July 2002. ftp://ftp-sop.inria.fr/oasis/publications/2002/josephVariamparambilStage2002.pdf.

[165] Philip Wadler. Listlessness is better than laziness II : composing listless functions. In *Workshop on Programs as Data Objects*, volume 217 of *LNCS*, Copenhagen, Denmark, October 1985. Springer-Verlag.

[166] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF : An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12) :31–37, 1994.